Pavel Töpfer

# ALGORITMY a programovací techniky

PROMETHEUS

Zpracoval RNDr. Pavel Töpfer, CSc.
Recenzovali Ing. Božena Mannová, M. Math.,
RNDr. Antonín Vrba, CSc.

Schválilo ministerstvo školství, mládeže a tělovýchovy ČR
dne 13. února 1995 pod č.j. 12153/95–23 k zařazení
do seznamu učebnic pro gymnázia a další střední školy

© Pavel Töpfer, 1995

**ISBN 80-85849-83-6**

## OBSAH

# 1 ÚVOD

Tato kniha je učebnicí základních datových struktur, algoritmů a programovacích technik. Některé postupy jsou pro řešení programátorských úloh typické a v různých obměnách se v programech často opakují. Právě takovéto obvyklé algoritmy a programátorské obraty vám kniha přiblíží a předvede jejich užití v konkrétních příkladech. Bude jen na vás, nakolik vás tyto příklady inspirují a usnadní vám tvorbu vašich vlastních programů.

Pro řešení jednoho zadaného problému můžeme použít různé algoritmy a zvolený postup řešení můžeme ještě různými způsoby naprogramovat. Dobrý programátor se liší od špatného kromě jiného tím, jak se staví k výběru algoritmu. Nespokojí se s prvním nápadem, jak by bylo možné danou úlohu řešit, ale porovnává různé postupy řešení a pečlivě z nich vybírá ten, který mu připadá nejvhodnější. Schopnost správné volby je jednou z klíčových dovedností každého dobrého programátora. Kniha by vám měla být návodem i v této oblasti. Ukáže vám, jaká kritéria se používají pro porovnávání algoritmů a programů, a na několika příkladech uvidíte rozdíl mezi „lepším“ a „horším“ řešením téže úlohy. Ukáže vám také některé základní techniky, jak navrhovat efektivní algoritmy.

Celá kniha je tematicky členěna tak, že každá kapitola je věnována jedné algoritmické nebo programátorské technice. Kapitoly na sebe bezprostředně nenavazují a můžete je číst v libovolném pořadí, které vám vyhovuje. Předem byste si však měli alespoň orientačně přečíst úvodní kapitoly 2.–7. věnované zavedení dále používaných základních matematických pojmů, informacím o uložení čísel v počítači, základním datovým strukturám a obecné problematice složitosti algoritmů. Pořadí dalších kapitol knihy ovšem také není zvoleno náhodně. V některých kapitolách se totiž objevují odkazy na algoritmy a na příklady z jiných částí knihy, neboť řada konkrétních úloh nám dobře poslouží k ilustraci více různých jevů, a patřila by proto vlastně na více míst zároveň. Budete-li číst knihu popořadě podle zvoleného uspořádání kapitol, nebudete muset listovat v knize směrem dopředu do „neznáma“, narazíte pouze na odkazy zpět do již přečtených kapitol.

V každé kapitole je nejprve úvodní výklad zvolené problematiky; ta je pak blíže objasněna na několika řešených příkladech. Myšlenky a principy všech použitých algoritmů a programů jsou vysvětleny s dosta-

tečnou podrobností slovně, aby byla kniha srozumitelná i bez detailního procházení programových ukázek. Některé algoritmy jsou dovedeny až do podoby vzorového řešení v jazyce Pascal, často i s upozorněním na výhodné obraty použité v programové realizaci nebo naopak na nedostatky uvedeného programu. Jednotlivé tematické celky jsou zakončeny zadáním dalších úloh v podobě cvičení.

V knize je pro zápis programových ukázek používán programovací jazyk Pascal. Všechny programy jsou odladěny na počítači typu PC v Turbo Pascalu, ale z důvodu snadné přenositelnosti a všeobecné srozumitelnosti jsou úmyslně psány tak, aby se v nich nevyužívala žádná rozšíření jazyka Turbo Pascal proti normě Pascalu. Jedinou naprosto nepodstatnou odchylkou od normy jazyka Pascal je to, že pro jednoduchost v hlavičkách programů neuvádíme jména souborů, s nimiž program pracuje, tj. zpravidla (input, output), jak to vyžaduje norma. Celý výklad je ovšem veden nezávisle na programovacím jazyku, takže detailní znalost právě Pascalu není pro čtení knihy nutná. Předpokládáme však, že čtenář zná alespoň v hrubých rysech buď Pascal, nebo některý jiný programovací jazyk podobného typu. Předpokládáme také, že čtenář již má alespoň minimální dřívější zkušenost s psaním vlastních jednoduchých programů nebo že se právě programovat učí.

Programové ukázky se v knize objevují ve dvou podobách. Většina z nich je zapsána ve tvaru úplného programu, některé mají tvar samostatné procedury nebo funkce. Celý program uvádíme tam, kde to má rozumný smysl, kde jde o řešení nějaké konkrétní úlohy jako celku s přesně definovanými vstupy a výstupy. Tvar procedury nebo funkce se objevuje v těch částech knihy, v nichž chceme předvést naprogramování určité technické operace, která se sice používá v různých programech, ale sama o sobě není řešením žádné reálné úlohy (např. manipulace se spojovými seznamy, vyhledávání hodnot v poli apod.). Výhodou úplných programů je to, že s nimi můžete bez dalších úprav ihned začít experimentovat na počítači, a tak se s vykládanou problematikou bezprostředně prakticky seznámit. Naleznete v nich přehledně a pohromadě deklarace všech potřebných datových struktur, realizaci vstupu dat, inicializace proměnných a po provedení vlastního výpočtu zase výstup výsledků. Až budete později psát své vlastní složitější programy, jistě se vám stane, že některý ze zde naprogramovaných algoritmů budete chtít využít k vyřešení nějaké dílčí podúlohy. V tom případě si již musíte sami přeměnit zde uvedený



program do podoby procedury nebo funkce s takovými parametry, jak to bude vyhovovat vaší úloze. Jádro algoritmu ovšem zůstane stejné, zachová se i způsob jeho naprogramování.

Učebnice je určena všem zájemcům o programování. Může být užitečná každému, kdo se učí programovat nebo kdo již zvládl základy programování, ale nemá dosud větší zkušenost s návrhem algoritmů a s posuzováním jejich kvality. Celá kniha je psána způsobem, který umožňuje samostatnou práci čtenáře s textem. Není však vhodná jako samostatné úvodní čtení pro úplné začátečníky. Její čtenář by měl být v programování alespoň mírně pokročilý, například na úrovni knihy [5]. Začátečník, který se programovat teprve učí, může s knihou pracovat pod odborným vedením učitele. Knihu lze využívat ve škole jako učebnici pro studenty i pro odbornou přípravu učitelů. Může také dobře posloužit jako zdroj rad a inspirací pro všechny, kdo vedou nejrůznější kursy programování, i jako studijní materiál pro posluchače takových kursů. Styl výkladu a obtížnost probírané látky odpovídá úrovni volitelné výuky programování na středních školách, povinné výuky ve specializovaných třídách gymnázií zaměřených na programování a základním kursům programování na vysokých školách technického a přírodovědného zaměření.

Některé příklady uvedené v knize pocházejí z našich vrcholných programátorských soutěží pro středoškoláky. Jsou vybrány z minulých ročníků kategorie P matematické olympiády a z korespondenčních seminářů z programování. Tyto soutěže se zaměřují právě na hledání takových řešení úloh, v nichž se nepoužije první nápad, ale kde hlubší úvaha vede autora programu k nalezení lepšího, efektivnějšího postupu řešení. Kniha proto dobře poslouží také pro přípravu nadaných studentů na účast v těchto i jiných programátorských soutěžích a pro individuální práci učitelů se špičkovými studenty.

Ačkoliv má kniha charakter učebnice pro výuku programování, není klasickou středoškolskou učebnicí, která obsahuje „vše potřebné“ a podle níž by mohla být systematicky vedena výuka ve stanoveném pořadí témat. Není to možné už proto, že se zde věnujeme pouze algoritmům a programovacím technikám, a ne výkladu programovacího jazyka. Dobrá výuka programování musí probíhat tak, aby se obě tyto stránky navzájem doplňovaly. Nemá smysl učit samotný programovací jazyk bez znalosti algoritmů a postupů ukazujících, jak se má jazyk využívat při psaní programů. Stejně tak není dobré učit samotné algoritmy bez znalostí

potřebných k jejich programové realizaci. Dnes je již k dispozici řada různých učebnic programovacích jazyků. Ty však obvykle vykládají pouze samotný programovací jazyk a jeho využití ukazují jen na velmi jednoduchých příkladech. Trochu obsažnější a přitom dostatečně srozumitelný výklad toho, jak postupovat při psaní programů, však již těžko někde najdete. Naše kniha se proto snaží alespoň částečně vyplnit tuto mezeru a poskytnout čtenáři souhrn základních znalostí z oblasti standardních datových struktur, algoritmů a programovacích technik, a to na úrovni dostupné i lepším studentům středních škol. Při výuce na škole by měla sloužit jako doplněk a protipól nějaké vhodné učebnice vyučovaného programovacího jazyka.

Samostatným doplňkem knihy je disketa obsahující všechny ukázkové programy použité v textu. Není na ní uloženo pouze několik zcela elementárních manipulací s poli a se spojovými seznamy. Text knihy nikde přímo neodkazuje na soubory uložené na disketě. Můžete s ním zcela bez problémů pracovat i v případě, že disketu s programy nevlastníte. Disketa je určena čtenářům, kteří chtějí do vykládané problematiky proniknout hlouběji a ukázkové programy si chtějí sami vyzkoušet spustit na počítači a experimentovat s nimi. V závěru knihy je seznam všech programových ukázek v pořadí, jak se objevují v knize. U každé ukázky je v závorce poznamenáno jméno souboru, v němž je příslušný program uložen. Přímo na disketě se nachází také textový soubor s obsahem celé diskety. Pod každým jménem souboru je stručně charakterizován jeho obsah a kapitola, kde se příslušný program objevuje v knize. Disketa obsahuje celkem 48 souborů zdrojového textu v Pascalu. Všechny programy jsou odladěny v prostředí Turbo Pascalu 6.0 a 7.0.

Jak jsme již uvedli, v některých případech není v knize otištěn celý program, ale pouze algoritmus zapsaný v podobě procedury nebo funkce. Pro potřeby ladění bylo ovšem nutné doplnit tyto procedury a funkce o nějaký malý hlavní program, v němž by se funkčnost podprogramů mohla ověřit. Soubor na disketě je pak zachován kompletní, tzn. včetně pomocného testovacího hlavního programu. Obsah každého souboru tudíž můžete ihned bez dalších úprav přeložit a spustit.

U několika programů je zajímavé sledovat, jak dlouho trvá výpočet pro různá vstupní data. U těchto programů jsme v souboru zachovali i volání příslušné funkce Turbo Pascalu, která přebírá od operačního



systému hodnotu času, a také zobrazení získaného systémového času (`KUN_BACK.PAS`, `KUN_HEUR.PAS`). V některých případech bylo pro potřeby ladění a testování výhodné mít více variant řešení téže úlohy pohromadě, tzn. více procedur v jednom souboru. I tento stav je zde zachován. Při testování pak stačí v pomocném hlavním programu zaměňovat jméno volané procedury (`BINSTROM.PAS`, `FIBONAC.PAS`, `CESTY.PAS`). Zvláštním případem je pak rozsáhlý soubor `NOTACE.PAS`, který obsahuje ve tvaru procedur a funkcí všechny operace s aritmetickými výrazy, jako jsou převádění mezi notacemi a vyhodnocování výrazu zapsaného v různých notacích. Soubor je sice na první pohled poněkud nepřehledný, drobnými zásahy do hlavního programu však umožňuje velmi výhodně testovat všechny tyto spolu těsně související podprogramy.

## 2 NĚKOLIK UŽITEČNÝCH POJMŮ

Na samém začátku knihy bude dobré připomenout si několik základních matematických pojmů, s nimiž se budeme setkávat později v textu. Omezíme se jen na to zcela nejnutnější a celý výklad povedeme spíše intuitivně, bez jakýchkoli matematických formalismů. Pokud pro vás bude tato krátká kapitola opakováním věcí, které již dávno znáte, je to jen a jen dobře a klidně ji při čtení přeskočte.

### 2.1 Množiny a posloupnosti

Úplná a přesná definice matematického pojmu **množina** je poměrně složitá a do této knihy rozhodně nepatří. Z našeho pohledu je však důležité vědět, že množinou rozumíme jakýkoli souhrn elementů (předmětů, čísel, údajů, jevů apod.), které z nějakého hlediska patří k sobě a které pak nazýváme prvky množiny. Počet prvků množiny není obecně omezen, může být konečný, nebo nekonečný. Množina neobsahující ani jeden prvek se nazývá prázdná množina. Všechny prvky téže množiny jsou navzájem různé, v jedné množině se žádný prvek nemůže vyskytovat vícekrát. Nijak se nezachovává pořadí, v němž byly jednotlivé prvky do množiny zařazeny.

K základním množinovým operacím patří test, zda daný prvek náleží do množiny, výpočet průniku, sjednocení a rozdílu dvou množin. Průnikem množin A, B nazýváme množinu těch prvků, které náležejí současně do množin A i B. Sjednocení množin A, B je množina všech prvků patřících alespoň do jedné z množin A, B. Rozdílem množin A a B rozumíme množinu prvků, které patří do A a přitom nepatří do B. Množiny můžeme také porovnávat. Množina A je podmnožinou množiny B, pokud každý prvek množiny A je zároveň prvkem množiny B. Množiny A a B se sobě rovnají, jestliže A je podmnožinou B a zároveň B je podmnožinou A.

Při psaní programů často pracujeme s množinami údajů (čísel, záznamů), které potřebujeme během výpočtu uschovat a později nějak dále zpracovávat (např. vyhledávat v nich danou hodnotu). Tyto množiny jsou samozřejmě vždy konečné. Vedle výše uvedených základních množinových operací používáme v programech ještě další, pro množiny typické akce, jako je průchod všemi prvky množiny spojený se zpracováním každého



prvku, vyhledávání všech prvků dané vlastnosti či nalezení nejmenšího prvku v množině.

Množina není jedinou strukturou sloužící k ukládání údajů v programech. Jinou často užívanou datovou strukturou je **posloupnost**. Obdobně jako množina je také posloupnost tvořena jistým souborem prvků, který může být konečný, nekonečný, nebo i prázdný. Na rozdíl od množiny je však pevně definováno pořadí prvků zařazených do posloupnosti, změnou tohoto uspořádání dostaneme odlišnou posloupnost. Navíc se v jedné posloupnosti mohou vícekrát opakovat prvky stejné hodnoty. Například (10, 20, 30, 20) a (20, 20, 30, 10) jsou dvě různé posloupnosti celých čísel.

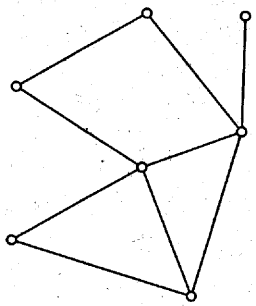
V programech budeme pracovat pouze s konečnými posloupnostmi. Operace s nimi prováděné jsou obdobné jako v případě množin. Budeme opět potřebovat testovat, zda je nějaký prvek v posloupnosti obsažen, budeme chtít vyhledat konkrétní prvek nebo nalézt všechny prvky dané vlastnosti. Musíme také umět projít celou posloupnost a s každým prvkem provést nějakou zvolenou akci. Na rozdíl od množin má zde navíc smysl ptát se na počet výskytů dané hodnoty v posloupnosti, na první a na poslední prvek posloupnosti.

### 2.2 Stromy a grafy

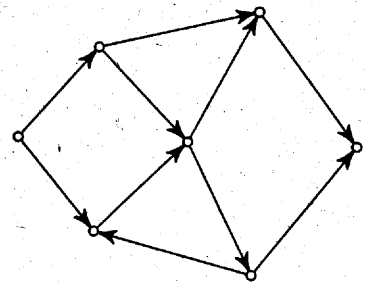
Slovo **graf** se v matematice používá ve dvou zcela odlišných významech. Prvním z nich je „graf funkce", tj. grafické vyjádření průběhu nějaké matematické funkce. Tento význam slova graf nás nyní zajímat nebude. Druhým významem slova graf, o který nám teď půjde, je velmi zhruba řečeno soustava bodů (tzv. uzlů nebo také vrcholů grafu), mezi nimiž vedou spojnice (tzv. hrany grafu). V matematice existuje samostatná disciplína zvaná teorie grafů, která se právě těmito grafy zabývá a studuje jejich vlastnosti. Pojem graf v ní je samozřejmě definován mnohem přesněji pomocí jiných matematických pojmů (viz např. [2], [15]). Pro naše potřeby se omezíme na tzv. konečné grafy, tj. budeme předpokládat, že množina vrcholů grafu je konečná. „Obrázek" tvořený vrcholy a mezi nimi vedoucími hranami grafu se pak správně nazývá nakreslení grafu. Tentýž graf má řadu různých nakreslení. Nakreslení grafu proto není jeho podstatnou charakteristikou, rozhodující je jen počet vrcholů grafu a které dvojice z nich jsou spojeny hranou. My se nebudeme zabývat teoretickými pojmy z teorie grafů a ke grafům budeme přistupovat spíše

intuitivně z hlediska jejich praktického využití v programování při řešení úloh. Jako dobrý příklad grafu z praxe nám může posloužit silniční síť vybudovaná mezi městy. Daných $N$ měst představuje vrcholy grafu. Mezi některými dvojicemi měst vedou přímé silnice — hrany grafu.

Budeme rozlišovat několik základních druhů grafů. Nejjednodušší grafy jsou takové, o nichž jsme dosud hovořili. Říká se jim obyčejné nebo také **neorientované grafy**. Naproti tomu existují **grafy orientované**. V nich jsou hrany představovány uspořádanými dvojicemi vrcholů. Mají tedy přiřazenu orientaci, z kterého do kterého vrcholu vedou. V nakreslení grafu tuto skutečnost vyznačujeme šipkou u každé hrany. V našem modelu silniční sítě si orientaci snadno zavedeme tak, že všechny silnice prohlásíme za jednosměrné. Orientované grafy představují obecnější prostředek než grafy neorientované. Neorientovaný graf totiž snadno nahradíme takovým orientovaným grafem se stejnou množinou vrcholů, v němž místo každé neorientované hrany mezi vrcholy $x$, $y$ vedou dvě hrany orientované, jedna z vrcholu $x$ do vrcholu $y$ a druhá naopak z vrcholu $y$ do vrcholu $x$. Při řešení konkrétních úloh se využívají někdy grafy orientované a někdy neorientované, podle charakteru úlohy. Budeme se proto věnovat oběma typům grafů.



*Obr. 1* Neorientovaný graf

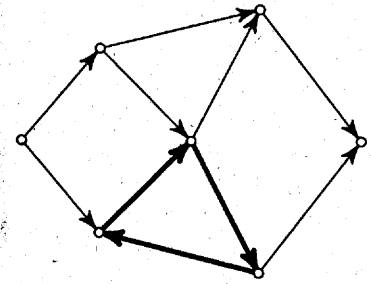


*Obr. 2* Orientovaný graf

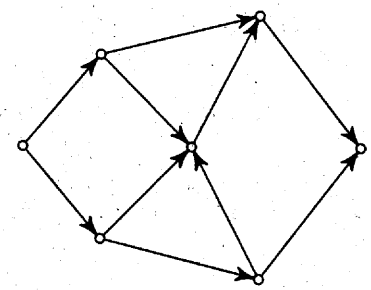
Dalším druhem grafů, který nás bude zajímat, jsou **grafy ohodnocené**. Ohodnocené grafy mohou být buď neorientované, nebo orientované. V ohodnoceném grafu je každé hraně přiřazeno tzv. **ohodnocení**, což může být například jedno celé číslo, ale také třeba číslo reálné, dvojice čísel apod. Nejjednodušším příkladem ohodnocení hran v modelu silniční sítě



délka silnice v kilometrech. Pokud by ale po silnicích jezdily autobusové oje, mohlo by nás zajímat jiné ohodnocení hran — silnic: každé hraně chom jako ohodnocení přiřadili dvojici čísel udávající dobu jízdy autousu v minutách a cenu jízdného v korunách. Sami jistě přijdete na řadu ných příkladů ohodnocení. Při práci s grafy budeme potřebovat ještě den důležitý pojem. **Cestou** z vrcholu $x$ do vrcholu $y$ rozumíme takovou osloupnost hran $h_1, h_2, h_3, \ldots, h_K$, že hrana $h_1$ vychází z vrcholu $x$, de do nějakého vrcholu, z něhož vychází hrana $h_2$, ta vede do vrcholu, ze erého vychází $h_3$, atd., až poslední hrana cesty $h_K$ vede do vrcholu $y$. příkladu se silniční sítí odpovídá cestě v grafu jednoduše cesta po stujících silnicích z jednoho města do druhého. Cesty v grafu nás zamají v případě neorientovaných i orientovaných grafů. U orientovaného rafu musí samozřejmě každá cesta postupovat pouze ve směru orientace an. Obvykle se předpokládá, že se na jedné cestě žádný vrchol grafu opakuje. Cesta, která vychází z nějakého vrcholu $x$ a vrací se zpět do hož vrcholu $x$, se nazývá **cyklus**. Orientovaný graf neobsahující žádný klus označujeme jako **graf acyklický**. Například graf znázorněný na br. 3 není acyklický, kdežto graf na obr. 4 ano.



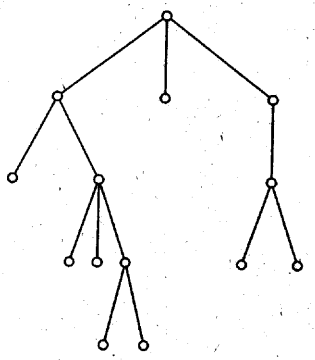
*Obr. 3* Cyklus v orientovaném grafu



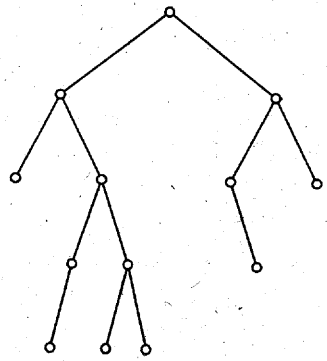
*Obr. 4* Acyklický graf

Pojem **strom** je v teorii grafů formálně definován jako zvláštní případ afu bez cyklů. Zvláštní skupinu mezi stromy pak tvoří tzv. **zakořeněné romy**, v nichž je jeden význačný vrchol označen jako **kořen stromu**. y zde opět vystačíme jen s intuitivní představou a v dalším textu se nezíme pouze na zakořeněné stromy, jejichž použití je pro programování

typické. Stromem budeme rozumět útvar, který je složen z jednotlivých vrcholů (uzlů) a který má následující tvar. Jeden význačný vrchol je kořenem stromu. Ten má několik **následníků** (někdy se také říká **synů**), každý z nich má své následníky, ti zase svoje atd. Přitom žádný vrchol není následníkem dvou nebo více jiných vrcholů zároveň. Naopak, každý vrchol s výjimkou kořene má právě jednoho předchůdce (svého „otce"). Některé vrcholy nemají již žádné své následníky a nazývají se **listy**. Výška stromu, tj. počet jeho hladin, je rovna maximální vzdálenosti nějakého listu od kořene. Teoreticky není nijak omezena, v programech ovšem pracujeme pouze s konečnými stromy. Zvláštním případem stromu je **strom binární**, v němž může mít každý vrchol nejvýše dva následníky. S binárním stromem se v programování setkáváme častěji než se stromem obecným. Slouží například pro reprezentaci aritmetického výrazu s binárními operátory nebo pro uložení dat do binárního vyhledávacího stromu či do haldy.



*Obr. 5* Strom



*Obr. 6* Binární strom

### 2.3 Logaritmus

Dalším matematickým pojmem, se kterým se v programování setkáte, je logaritmus. Budeme ho často používat při odhadování složitosti výpočtu. Logaritmická funkce je v matematice zavedena jako funkce inverzní k funkci exponenciální. Říkáme, že $Y$ je logaritmem čísla $X$ při základu $Z$ (píšeme $Y = \log_Z X$) právě tehdy, jestliže platí $X = Z^Y$. Konstanta $Z$ představující základ logaritmu je kladné reálné číslo různé od 1. Při numerických výpočtech se dříve běžně pracovalo s dekadickými logaritmy ($Z = 10$), ve vyšší matematice se využívají výhodné vlastnosti logaritmů přirozených (jejich základem je Eulerova konstanta e = 2,718 281 82...). V programování se setkáváme nejčastěji s logaritmy o základu 2. Objevují se ve všech úvahách o rychlosti výpočtu takových algoritmů, které jsou založeny na nějakém půlení intervalů nebo na využití binárních stromů.

Zdržíme se ještě chvíli u binárních stromů a podíváme se na vztah mezi počtem uzlů a výškou binárního stromu. Mnohé algoritmy jsou totiž založeny na výhodném uložení potřebných údajů právě do struktury binárního stromu, díky čemuž jsou pak časové nároky příslušného algoritmu úměrné výšce tohoto stromu. Binární strom s $N$ uzly může mít v nejhorším případě výšku až $N$. Tento případ nastane tehdy, je-li strom degenerovaný, tj. každý jeho uzel má jenom jednoho následníka. Nejpříznivější je naopak případ, když je strom tzv. vyvážený a každý uzel s výjimkou uzlů v posledních dvou hladinách má oba dva syny. Výška takového binárního stromu je nejmenší možná pro daný počet uzlů $N$ a je rovna přibližně $\log_2 N$, jak si hned ukážeme.

Úplný binární strom se všemi hladinami plně obsazenými má na první hladině jeden uzel, na druhé dva, na třetí čtyři, na čtvrté osm atd. Obecně na $K$-té hladině má $2^{K-1}$ uzlů. Binární strom s nejmenší možnou výškou má na všech hladinách od první, tj. od kořene, až do předposlední hladiny plný počet uzlů $2^{K-1}$ (kde $K$ je pořadové číslo hladiny), zbývající uzly jsou v poslední hladině. Na poslední $H$-tou hladinu připadne nejméně jeden a nejvýše všech $2^{H-1}$ uzlů. Takový strom o $H$ hladinách má tedy nejméně $1+2+\ldots+2^{H-2}+1 = 2^{H-1}$ uzlů a nejvýše $1+2+\ldots+2^{H-1} = 2^H - 1$ uzlů. Označíme-li počet uzlů $N$, platí nerovnosti

$$2^{H-1} \leqq N \quad \text{a} \quad N \leqq 2^H - 1.$$

Odtud zlogaritmováním dostaneme přípustné rozmezí pro počet hladin

$$H \leqq \log_2 N + 1 \quad \text{a} \quad H \geqq \log_2(N+1).$$

Oba výrazy $\log_2 N + 1$ a $\log_2(N+1)$ se příliš neliší od $\log_2 N$. Můžeme tedy také říci, že výška binárního stromu $H$ je přibližně rovna $\log_2 N$.

# 3 SLOŽITOST ALGORITMŮ

Máme-li řešit nějakou programátorskou úlohu, napadne nás obvykle více různých postupů řešení. Z nich si musíme vybrat jeden, na němž založíme náš program. Je proto třeba stanovit nějaká kritéria pro rozhodování o tom, který postup vede k programu lepšímu a který k horšímu, co to vlastně znamená „lepší" a „horší". Pro posuzování kvality algoritmů existuje řada možných hledisek. Nejprimitivnějším kritériem může být například volba toho postupu, který nás napadne jako první nebo který se nám bude nejsnadněji programovat, bez ohledu na vlastnosti výsledného programu. Ani takovýto jednoduchý způsob výběru algoritmu nelze předem odmítnout a v některých konkrétních situacích může být praktický a užitečný. Obvykle však posuzujeme algoritmy jinak, na základě vlastností výsledného programu. Budou nás zajímat časové a paměťové nároky programu, tzn. rychlost výpočtu a velikost potřebné operační paměti počítače. Těmto vlastnostem algoritmů a programů říkáme **časová** a **paměťová** (nebo také prostorová) **složitost**. Vedle pojmu složitost se setkáte také s označením **efektivita algoritmu**. Pojmy složitost a efektivita jsou vlastně doplňkové a charakterizují tutéž vlastnost algoritmu. Tvrzení „algoritmus A má větší časovou složitost než algoritmus B" znamená totéž jako věta „algoritmus A má menší časovou efektivitu než algoritmus B" nebo také „algoritmus A je časově méně efektivní než algoritmus B".

Problematika efektivity algoritmů je poměrně náročná a má bohaté matematické zázemí. V naší knize však nebudeme příliš zabíhat do teoretických detailů. Kniha není a nechce být učebnicí teorie výpočetní složitosti, a proto se v zájmu srozumitelnosti textu záměrně vyhneme exaktním matematickým odvozením a důkazům. Omezíme se pouze na zavedení základních pojmů a na objasnění praktického významu, proč se otázkami složitosti algoritmů musíme při programování zabývat. Odhady efektivity konkrétních algoritmů budou provázet všechny další kapitoly této knihy.



## 3.1 Význam efektivity

Prozatím jsme uvedli dvě pro nás významná hlediska hodnocení algoritmů, a sice složitost časovou a paměťovou. Jde o dvě odlišná kritéria, která při posuzování konkrétních algoritmů stojí velmi často proti sobě. Mnohdy máme k dispozici dva postupy řešení, z nichž jeden je pomalejší a vystačí s velmi malou pracovní pamětí, zatímco druhý je rychlejší, ale má vyšší paměťové nároky. Stává se totiž, že ke zrychlení výpočtu musíme použít nějakou pomocnou datovou strukturu, v níž budou uchovávány předem spočtené a připravené hodnoty nebo mezivýsledky. Nejrychlejší algoritmus tudíž nebývá optimální z hlediska paměťové složitosti a naopak algoritmus s nejmenšími paměťovými nároky zase nebývá nejrychlejší. V takovém případě se musíme rozhodnout, kterému hledisku dáme při volbě algoritmu přednost. Na tuto otázku neexistuje jednoznačná odpověď, záleží vždy na konkrétním řešeném problému a na podmínkách, jaké pro jeho řešení máme (například na požadavcích zadavatele úkolu, jak velkou operační paměť máme na počítači k dispozici nebo jak dlouho můžeme čekat na výsledky). V současné době se v programování nejčastěji staví na první místo efektivita časová, nevede-li tato volba k příliš extrémním nárokům na paměť. Tohoto pohledu se budeme držet i v této knize a při návrhu algoritmů a programů budeme usilovat o co nejmenší časovou složitost.

Ještě před zavedením všech potřebných pojmů si můžeme na jednom malém příkladu ze života ukázat, jaký praktický význam může mít volba vhodného algoritmu. Pravděpodobně jste již vícekrát v životě hledali nějaké telefonní číslo v telefonním seznamu. Při hledání telefonního čísla můžeme postupovat různě. Nejjednodušší by bylo procházet celý telefonní seznam od začátku postupně jméno po jméně, dokud nenajdeme to hledané. V programování se tato metoda nazývá sekvenční vyhledávání. Nalézt telefonní číslo pana Adámka bychom tímto způsobem ještě zvládli, ale v případě pana Žežulky by bylo takovéto hledání asi nad lidské síly. Ve skutečnosti, jak sami dobře víte, hledáme v telefonním seznamu úplně jinak. Všemi intuitivně používaný mnohem rychlejší postup je založen na myšlence půlení intervalů. Využíváme přitom skutečnosti, že jsou jména telefonních účastníků v seznamu utříděna podle abecedy. V závěru kapitoly se k této úloze ještě vrátíme a oba postupy vyhledávání porovnáme přesněji.

Nyní si musíme objasnit ještě jednu základní vlastnost algoritmů.

Doba výpočtu a paměťové nároky nebývají v případě většiny algoritmů neměnné, ale nějak závisejí na vstupních datech. Algoritmus zpravidla slouží k řešení celé třídy podobných problémů lišících se právě zadáním vstupních údajů. Délku trvání výpočtu a potřebnou velikost paměti obvykle neovlivňují konkrétní zadané hodnoty, ale jen velikost vstupních dat. Přesná definice pojmu velikost vstupních dat by byla dosti obtížná, do značné míry závisí na řešeném problému. Zpravidla se jedná o počet čísel nebo znaků, které algoritmus zpracovává. Například doba výpočtu algoritmu na nalezení maxima z daných $N$ čísel závisí právě na počtu čísel $N$ (ale již ne na jejich konkrétních hodnotách, vždy se provede přesně $N-1$ porovnání). Rychlost algoritmu na násobení dvou čtvercových matic typu $N \times N$ je dána rozměrem matic $N$. Řadu dalších příkladů uvedeme později.

V některých méně častých případech mohou časové a paměťové nároky algoritmu záviset i na hodnotách vstupních dat. Tato závislost může být různého typu a je třeba řešit ji případ od případu. Jedním takovým příkladem je situace, že algoritmus pracuje s tak velkými vstupními hodnotami, že s nimi nemůže nakládat jako s elementárními číselnými údaji. Musí si proto sám vytvářet nějakou jejich nestandardní vnitřní reprezentaci a například vlastní vícenásobnou aritmetiku. Zde se hodí vzít za měřítko velikosti vstupních dat nikoliv počet takových dlouhých čísel, ale počet jejich číslic (neboť algoritmus je musí zpracovávat po znacích). Jinou situaci představují algoritmy generující rozsáhlé výstupy, jejichž velikost přímo závisí na vstupním údaji a je přitom určující pro dobu trvání výpočtu (například úloha vypsat všechna čísla jisté vlastnosti menší než dané $N$). V takovém případě nebudeme při určování složitosti algoritmu brát v úvahu velikost vstupních dat (v našem příkladě je to třeba jen jediné číslo $N$), ale buď přímo zadanou hodnotu, nebo velikost výstupních dat. Velikost některých problémů je přirozeným způsobem charakterizována několika parametry. Například rychlost algoritmu pracujícího s obdélníkovými maticemi tvaru $N \times M$ bude záviset na obou rozměrech matic $N$ a $M$. Složitost grafových algoritmů (viz kap. 9) zase obvykle vyjadřujeme s ohledem na počet vrcholů a počet hran zkoumaného grafu.



## 3.2 Časová a paměťová složitost

Časovou složitostí algoritmu rozumíme závislost jeho časových nároků na velikosti konkrétního řešeného problému nebo konkrétních vstupních dat. Má-li to být charakteristika algoritmu, nemůžeme ji měřit v běžných časových jednotkách. Skutečná doba výpočtu programu na počítači vyjádřená třeba v sekundách závisí nejen na algoritmu, ale také na použitém programovacím jazyce, kvalitě překladače, rychlosti procesoru počítače apod. Časovou složitost proto určujeme pouze počtem elementárních operací (kroků algoritmu), které budou provedeny při výpočtu programu s danými vstupními daty. Je to tedy funkce, která každé hodnotě $N$ udávající velikost konkrétního řešeného problému přiřazuje počet operací vykonaných při výpočtu podle daného algoritmu. Tato funkce je zpravidla rostoucí.

Paměťovou složitost algoritmu definujeme podobně jako závislost paměťových nároků algoritmu na velikosti řešeného problému nebo vstupních dat. Měříme ji v běžných jednotkách velikosti paměti, tedy v bitech nebo bytech, pro jednoduchost ji často udáváme počtem jednoduchých proměnných, které budou při výpočtu podle algoritmu zapotřebí. Paměťová složitost je tedy opět funkce, která každé velikosti vstupních dat $N$ přiřazuje počet paměťových míst potřebných pro uskutečnění výpočtu.

Všechny další pojmy se zavádějí stejným způsobem pro časovou pro paměťovou složitost algoritmů. Omezíme se proto v dalším výkladu pouze na časovou složitost, která pro nás ostatně bude, jak jsme již uvedli dříve, hlavním kritériem pro posuzování kvality algoritmů. Zavedení všech dalších pojmů pro paměťovou složitost by bylo obdobné.

Při studiu algoritmů budeme rozlišovat časovou složitost v nejhorším případě a časovou složitost v průměrném případě. Časová složitost z hlediska **nejhoršího případu** udává pro každé $N$, jak nejdéle může trvat výpočet podle algoritmu s libovolnými vstupními daty velikosti $N$. Má tedy význam zaručené horní meze, kterou doba výpočtu s daty velikosti $N$ rozhodně nepřekročí. Analýzou algoritmu obvykle celkem snadno určíme buď přímo jeho časovou složitost v nejhorším případě, nebo alespoň hrubší horní odhad této složitosti. Chceme-li dobře porovnávat algoritmy, musíme si dávat dobrý pozor na případnou nepřesnost takového hrubého horního odhadu a vždy se musíme snažit nalézt skutečnou časovou složitost. Pokud například v programu vidíme na první pohled tři do sebe vnořené cykly, z nichž každý bude rozhodně proveden nejvýše $N$-krát (ale

možná i méněkrát), je velmi snadné říci, že takový program má horní odhad časové složitosti $N^3$. Po detailnější analýze ale možná zjistíme, že příkazy obsažené ve vnitřním cyklu budou provedeny vždy nejvýše $N^2$-krát, a že má tedy tento algoritmus ve skutečnosti časovou složitost pouze $N^2$. (Příklad takového programu je uveden v kap. 9.2.)

Časová složitost z hlediska **průměrného případu** určuje pro každé $N$ průměrnou délku výpočtu při zadání dat velikosti $N$. Má tedy význam doby, kterou můžeme v průměru očekávat, že bude k výpočtu zapotřebí. Z hlediska praktického použití algoritmu je tento údaj velmi zajímavý a užitečný, většinou lépe charakterizuje algoritmus než složitost v nejhorším případě. Bývá však dosti obtížné průměrnou časovou složitost algoritmu určit. Při jejím odvození je nutné uvažovat pravděpodobnostní rozložení všech možných vstupních dat zvolené velikosti. Obvykle se proto při posuzování algoritmů pracuje se složitostí z hlediska nejhoršího případu. Také my v této knize budeme nadále používat, aniž bychom to stále připomínali, pouze složitost algoritmů v nejhorším případě.

U některých algoritmů je přímo nezbytné nahradit přesný vzorec vyjadřující časovou složitost algoritmu vhodným odhadem (zpravidla horním). Představte si jednoduchý algoritmus, který zjišťuje, zda je dané číslo $N$ prvočíslo. Postupuje tak, že zkouší dělit $N$ postupně všemi celými čísly od 2 až do druhé odmocniny z $N$. Jestliže žádného dělitele čísla $N$ nenajde, označí $N$ za prvočíslo. Pokud nějakého dělitele najde, $N$ není prvočíslo a výpočet ihned skončí. Počet vykonaných operací je velmi závislý na konkrétní hodnotě $N$. Například pro každé sudé $N$ skončí výpočet již po prvním kroku výpočtu nalezením dělitele 2, zatímco pro prvočíselná $N$ se provede plný počet testů rovný odmocnině z $N$. Funkce, která by vyjadřovala přesnou závislost počtu provedených operací na hodnotě $N$, je v tomto případě velmi komplikovaná a těžko bychom nalezli nějaké její exaktní matematické vyjádření. Samozřejmě není ani rostoucí. Chceme-li tedy umět něco rozumného říci o časové složitosti uvedeného algoritmu, musíme se omezit na nalezení co nejlepšího horního odhadu této „divoké" funkce. Takovým odhadem je funkce druhá odmocnina z $N$.



## 2.3 Porovnávání algoritmů

Nyní je třeba rozhodnout, který z algoritmů s různou časovou složitostí je lepší, tzn. který z nich povede k rychlejšímu programu. Časová složitost je neklesající funkce a pro dvě takové funkce vůbec nemusí platit, že by jedna měla menší funkční hodnoty než druhá v celém definičním oboru. To můžeme snadno ilustrovat na jednoduchém příkladu: pro lineární funkci $f(n) = 10n$ a kvadratickou funkci $g(n) = n^2$ platí vztah $f(n) > g(n)$ pro $n < 10$ a naopak $f(n) < g(n)$ pro $n > 10$. Máme-li tedy v praxi porovnat dva algoritmy podle jejich časové složitosti a zvolit jeden z nich k naprogramování řešené úlohy, bude záležet i na tom, zda v zadání úkolu vymezeno, pro jak velká vstupní data bude program používán. Pokud takové vymezení nemáme, bude pro nás podstatné, který algoritmus je rychlejší pro velké hodnoty $N$ (kde $N$ je velikost vstupních dat, resp. velikost řešené úlohy). Mluvíme pak o **asymptotické časové složitosti** a ze dvou algoritmů prohlásíme za lepší ten, který má asymptotickou složitost menší. Je to tedy ten, jehož funkce časové složitosti roste pomaleji s rostoucí hodnotou $N$. Není totiž zpravidla příliš rozumné, který algoritmus je rychlejší pro malá $N$, neboť rozsahem malý problém lze vyřešit v rozumně krátkém čase i horším algoritmem.

Při odvozování asymptotické složitosti algoritmů je pro nás podstatná pouze rychlost růstu příslušné funkce. Není důležité znát přesný počet provedených elementárních operací a bylo by také velmi pracné a obtížné hledat vzorec, který by určoval počet operací detailně. Tak algoritmu, jehož časovou složitost charakterizuje předpis $2N^2 + 3N +$ [illegible] stačí zjistit, že s rostoucím $N$ jsou jeho časové nároky úměrné $N^2$ neboli že příslušná funkce časové složitosti je kvadratická. Říkáme také, že takový algoritmus má kvadratickou časovou složitost. Tuto skutečnost bývá [illegible] zapisovat ve tvaru $O(N^2)$ s použitím zvláštního matematického symbolu „velké $O$". My budeme takovýto zápis rovněž používat. Pokud vám matematická definice symbolu $O$ uvedená v závěru kapitoly nebyla dostatečně srozumitelná, můžete ji vynechat a chápat zápisy typu $O(N^2)$ intuitivně v právě uvedeném významu.

Má-li přesné vyjádření funkce časové složitosti algoritmu tvar součtu, pak rychlost růstu této funkce je určována rychlostí růstu nejrychleji rostoucího členu. Například v případě polynomu je to člen s nejvyšší mocninou $N$. Uměle je možné vytvářet algoritmy s nejrozmanitějšími časovými složitostmi, ale v praxi používané algoritmy mívají většinou

některou z následujících složitostí: $O(\log N)$, $O(N)$, $O(N \log N)$, $O(N^2)$, $O(N^2 \log N)$, $O(N^3)$, ..., $O(2^N)$. Přitom stupeň polynomu bývá poměrně nízký. Uvedené typické asymptotické složitosti algoritmů jsou uspořádány vzestupně podle rychlosti růstu. Algoritmům se složitosti $O(N)$ říkáme lineární, se složitosti $O(N^2)$ kvadratické, se složitostí $O(N^3)$ kubické. Všechny algoritmy, jejichž funkci časové složitosti můžeme shora omezit polynomem v $N$, označujeme jako algoritmy **polynomiální**. Algoritmy s větší časovou složitostí nazýváme **exponenciální**. Jejich typická složitost je $O(2^N)$.

Skutečnost, že se při určování asymptotické časové složitosti omezujeme pouze na řádový odhad růstu příslušné funkce, má pro nás ještě jeden význam. Nejenže nalezení takového odhadu je mnohem snadnější, než by bylo hledání přesného předpisu, ale vyhneme se tím do značné míry problému s přesným určováním, jaké instrukce vlastně máme při vyjadřování časové složitosti počítat. Řekli jsme, že časová složitost se měří počtem provedených elementárních instrukcí při výpočtu, ale co jsou ony elementární instrukce? Při zápisu algoritmu ve tvaru programu ve strojovém kódu nebo v assembleru bychom mohli celkem snadno prostě spočítat počet všech provedených příkazů (ale i zde je otázka, zda je to zcela správné, když ne každá instrukce strojového kódu je procesorem vykonána stejně rychle). U programu zapsaného ve vyšším programovacím jazyce jsou tyto problémy mnohem větší a řeší se obvykle tím, že se uvažují instrukce typické pro řešený problém (např. počet porovnání, přesunů v paměti, aritmetické operace apod.). Pro určení řádového odhadu složitosti to dobře postačuje, zatímco určení přesného předpisu funkce vyjadřujícího počet provedených operací by bylo obtížné (jak např. započítávat operace „schované" v řídících konstrukcích strukturovaných příkazů jazyka).

V některých případech se s řádovým odhadem asymptotické složitosti algoritmu nespokojíme. Pokud budeme například vybírat mezi dvěma různými algoritmy s lineární časovou složitostí, bude pro naši volbu podstatné, že jeden z nich potřebuje ke zpracování dat velikosti $N$ čas přibližně $2N$ a druhý $4N$. První z nich je zřejmě dvakrát rychlejší. Mezi více algoritmy se stejnou řádovou rychlostí růstu asymptotické časové složitosti volíme ten lepší na základě detailnější analýzy jejich složitosti. Obvykle se při tom omezujeme pouze na určení multiplikativní konstanty u členu rozhodujícího pro rychlost růstu. Pokud by se i tato konstanta



u obou porovnávaných algoritmů shodovala, algoritmy považujeme za stejně rychlé a vybereme z nich podle jiných kritérií (nebo třeba náhodně). Odlišnosti v pomaleji rostoucích členech se určují mnohem obtížněji a z hlediska celkových časových nároků výsledného programu jsou pro velká $N$ naprosto zanedbatelné.

Studium časové a paměťové složitosti algoritmů je významné z hlediska praktické použitelnosti algoritmů při řešení různých úloh. Programátor si musí stále uvědomovat, že jeho program bude počítat v omezené paměti počítače a v omezeném čase. Teoreticky správně fungující program je naprosto nepoužitelný, pokud se při výpočtu s reálnými vstupními daty nedočkáme výsledků nebo když výpočet zhavaruje pro přeplnění operační paměti počítače. Analýza složitosti algoritmů nám dává jasnou odpověď na laickou představu typu „program počítá pomalu, ale to vůbec nevadí, použití rychlejšího počítače vše napraví". Rychle zjistíme, že tomu tak rozhodně nemusí být. Zpracování malých dat obvykle nezpůsobuje žádné problémy, ale rozsáhlejší data mohou výpočet výrazně pozdržet. U programů, jejichž časová složitost je dána rychle rostoucí funkcí, se při prodlužování doby výpočtu nebo při přechodu na rychlejší počítač jen velmi pomalu zvětšuje rozsah dat, která je možné zpracovat ve stanoveném čase. V případě lineárního algoritmu se desetinásobné zrychlení procesoru (nebo prodloužení doby výpočtu) projeví desetinásobným zvětšením zpracovaných dat. U kvadratického algoritmu umožní toto zrychlení jenom zhruba trojnásobné zvětšení rozsahu dat. Má-li ale algoritmus exponenciální časovou složitost $2^N$, pak desetinásobné zrychlení nebo prodloužení výpočtu umožní zvětšit rozsah zpracovaných dat jen přibližně o 3.

Z hlediska praktické použitelnosti při práci s většími daty stojí výrazný předěl mezi algoritmy polynomiálními a algoritmy exponenciálními. Zatímco hodnota polynomu bývá i pro větší $N$ ještě přijatelná, a výpočet je tedy v principu časově zvládnutelný, rychlost růstu exponenciální funkce zcela vylučuje použít takový algoritmus k výpočtu již pro poměrně malé hodnoty $N$. Jestliže si sami zkusíte naprogramovat nějaký jednoduchý algoritmus s časovou složitostí $O(2^N)$ a budete ho testovat na běžném počítači typu PC, přesvědčíte se, že zpravidla již pro hodnoty $N$ kolem 50 se výsledku prostě nikdy nedočkáte. Jakékoliv reálně možné prodlužování doby výpočtu nebo zrychlování procesoru nám nemůže v žádném případě pomoci. Při návrhu algoritmů se proto

snažíme vyhýbat se exponenciálním algoritmům všude, kde je to jen trochu možné.

Rozhodně nám bude stát za to vybrat rychlejší ze dvou algoritmů i v případě, že se jejich asymptotické složitosti příliš neliší, ale předpokládáme práci s rozsáhlými daty. Jako příklad nám může posloužit snad nejznámější programátorská úloha — utřídění $N$ čísel podle velikosti. V literatuře lze nalézt celou řadu různých algoritmů, s některými z nich se seznámíme také později v této knize (v kap. 11). Jednodušší třídicí algoritmy mají časovou složitost $O(N^2)$ (například třídění přímým výběrem, přímým zatřiďováním nebo bublinkové třídění), zatímco ty lepší pracují v čase $O(N \log N)$ (např. třídění sléváním nebo třídění haldou). Porovnejme tedy dva třídicí algoritmy, z nichž jeden provede při setřídění $N$ čísel $N^2$ operací porovnání a druhý $N \log N$ porovnání. Pokud by na našem počítači trvalo jedno porovnání čísel 0,1 ms, pak utřídění 100 čísel pomocí prvního algoritmu by trvalo 1 sekundu a pomocí druhého algoritmu přibližně 0,07 s. To je sice řádový rozdíl, ale z hlediska výsledné doby výpočtu v podstatě nezajímavý. Sekunda je tak krátký čas, že i pomalejší algoritmus nám plně vyhovuje. Kdybychom však potřebovali utřídit 100 000 čísel (předpokládejme pro tuto chvíli, že se nám všech těchto 100 000 čísel vejde najednou do operační paměti počítače — ne na každém počítači to bude pravda), rozdíl mezi oběma algoritmy nabude zásadních rozměrů: program realizující kvadratický algoritmus by počítal více než 11 dní, zatímco program s časovou složitostí $N \log N$ zvládne stejnou práci za necelé 3 minuty.

Rozdíl mezi algoritmy s různou časovou složitostí si můžeme demonstrovat i na našem úvodním příkladu o vyhledávání v telefonním seznamu. Současný pražský telefonní seznam bytových telefonních stanic obsahuje více než 410 000 jmen. Pokud bychom v něm hledali zvolené jméno tím nejprimitivnějším způsobem, tj. postupným prohledáváním, a pokud bychom potřebovali na porovnání dvou jmen pouhou jednu sekundu (rychleji to člověk asi nedokáže), mohli bychom strávit nad telefonním seznamem při hledání jednoho čísla téměř 5 celých dní a nocí nepřetržité vysilující práce. Používáme totiž algoritmus sice s celkem příznivou lineární časovou složitostí, ale zato na zpracování velmi rozsáhlého souboru dat (a člověk je pomalý „procesor", jehož práci již nelze příliš urychlit). V praktickém životě postupujeme při hledání v telefonním seznamu odlišně. I bez znalosti jakékoli teorie intuitivně využíváme



princip binárního vyhledávání v uspořádané posloupnosti, který je založen na myšlence půlení intervalů. Otevřeme telefonní seznam uprostřed a nalezené jméno porovnáme s hledaným. Na základě tohoto porovnání zjistíme, v které polovině seznamu má smysl dále hledat. V této části seznamu potom pokračujeme v hledání stejným způsobem. V každém kroku tedy porovnáme jedno jméno ze seznamu s hledaným jménem a zmenšíme sledovanou část seznamu na polovinu. Po druhém kroku nás bude zajímat už jen čtvrtina telefonního seznamu, po třetím osmina seznamu, po čtvrtém jedna šestnáctina atd. Jestliže obsahuje celý seznam $N$ jmen, musíme vykonat přibližně $\log_2 N$ kroků, aby se zkoumaný úsek seznamu zúžil na jediné jméno. Binární vyhledávání má tedy časovou složitost $O(\log N)$. Jedno telefonní číslo díky tomu najdeme v seznamu při stejné rychlosti porovnávání jmen zhruba za 20 sekund.

### 1.4 Dodatky k otázce složitosti

Vedle složitosti algoritmu (resp. programu) zavádíme také pojem **složitost problému**. Tento pojem vychází z teoretické představy, že máme k dispozici všechny programy řešící daný problém a porovnáváme jejich složitost. Časová složitost problému je pak rovna časové složitosti nejrychlejšího z algoritmů, které problém řeší. Má tedy význam dolního odhadu složitosti, kterého lze dosáhnout. Říká nám, že v principu nemůže existovat algoritmus, který by řešil tento problém s menší složitostí. Stanovit složitost nějakého problému je velmi obtížný úkol. Nemůžeme samozřejmě posuzovat všechny různé algoritmy či programy řešící daný problém, těch je nekonečně mnoho. Odvození je třeba vést jinou cestou. My se zde touto problematikou nebudeme více zabývat a pro ilustraci jen uvedeme jeden příklad: Známá úloha uspořádat $N$ čísel podle velikosti (jestliže o zpracovávaných číslech nic bližšího nevíme) má časovou složitost $O(N \log N)$. Zdůvodnění tohoto výsledku naleznete v kap. 11.2.

Kapitolu o časové a paměťové složitosti uzavřeme slíbenou matematickou definicí symbolu $O$. Mějme dvě funkce $f$, $g$ definované v oboru přirozených čísel (v matematické analýze se totéž zavádí analogicky v oboru reálných čísel). Řekneme, že funkce $f$ je třídy $O(g)$, jestliže existuje taková kladná reálná konstanta $C$, že pro všechna přirozená čísla od jistého $n_0$ počínaje platí $f(n) \leqq Cg(n)$. To znamená, že funkce $g$ shora omezuje funkci $f$ až na multiplikativní konstantu. Vzhledem k tomuto

jednostrannému omezení může být vyjádření složitosti algoritmu pomocí symbolu $O$ dosti hrubé. Kvadratická funkce $2N^2 + 3N + 1$ je totiž třídy $O(N^2)$, ale podle uvedené definice patří také do třídy $O(N^3)$, $O(N^4)$ atd. Proto se někdy vedle symbolu $O$ zavádí ještě další symbol pro hodnocení složitosti algoritmů, symbol $\theta$. Řekneme, že funkce $f$ je třídy $\theta(g)$, jestliže $f$ je z $O(g)$ a zároveň $g$ je z $O(f)$. To znamená, že funkce $f, g$ rostou stejně rychle až na multiplikativní konstantu. Kvadratická funkce $2N^2+3N+1$ je třídy $\theta(N^2)$, ale nepatří do třídy $\theta(N^3)$.

Při skutečném srovnávání algoritmů bychom správně měli posuzovat jejich složitost podle tříd složitosti $\theta$, nikoli podle $O$. Analýzou algoritmu však obvykle dostáváme pouze horní odhad počtu provedených instrukcí nebo potřebných paměťových míst. Při pečlivě provedené analýze tento odhad zpravidla nebývá příliš hrubý a řádově se neodlišuje od skutečných nároků (je to tedy nejen určení třídy $O$, ale i třídy $\theta$). Dokazovat tuto skutečnost formálně však bývá pracné, bez důkazu by zase použití symbolu $\theta$ bylo nekorektní. Budeme proto nadále vyjadřovat složitost algoritmů pouze pomocí symbolu $O$, jak je tomu ostatně zvykem i v jiné, ne vysloveně odborné literatuře. Přitom vždy budeme usilovat o to, aby byl náš odhad asymptotické složitosti algoritmu co nejlepší.

## CVIČENÍ

**1.** Podle postupu uvedeného v závěru kap. 3.2 naprogramujte algoritmus na testování, zda je dané celé číslo prvočíslo. Sledujte počet porovnání, která se vykonají při výpočtu se vstupní hodnotou $N$ z rozmezí od 1 do 50. Nakreslete si graf závislosti počtu provedených porovnání na $N$.

**2.** Vylepšete algoritmus z cvičení 1 tak, aby se počet potřebných porovnání snížil. *Návod*: Odlište sudá a lichá $N$, sudé dělitele není třeba testovat.



## ...ENÍ ČÍSLO JAKO ČÍSLO

V programech zpracováváme údaje a informace různého druhu — ...avidla číselné, znakové a textové. Práce se znaky a texty většinou ...ůsobí žádné velké problémy. Tyto údaje se často pouze čtou ze vstupu, ...hovávají v datových strukturách a později tisknou, popř. se v nich něco ...ledává. Při výpočtech s číselnými hodnotami se však můžeme dočkat ...ých nemilých překvapení. Věnujeme proto tuto kapitolu krátké in...maci o tom, jak se v počítači ukládají a zpracovávají čísla.

### Celá a reálná čísla

Bývá zvykem rozlišovat v programech čísla „celá" a „reálná". Běžné ...gramovací jazyky včetně Pascalu nám přímo nabízejí pro tyto dvě ...egorie čísel různé předdefinované standardní datové typy. Celočíselný ...ový typ bývá označován nejčastěji jako `integer` nebo `int`, reálný ...elný typ nese zpravidla označení `real` nebo `float`. Někomu by mohlo ...padat poněkud nelogické, proč vůbec takto čísla rozlišovat, když každé ...é číslo je již zároveň číslo reálné. Důvodem tohoto rozdělení čísel je ...lišný způsob jejich vnitřní reprezentace a v souvislosti s tím i prová...ní výpočtů. V mnoha programech se objevují číselné proměnné, které ...hou nabývat jedině celých hodnot. Vyplatí se zavést si pro ně zvláštní ...očíselný typ, neboť práce s celými čísly bývá na počítačích jednodušší ...ychlejší než práce s čísly reálnými. Celá čísla se navíc ukládají přesně, ...ímco při jakékoli manipulaci s čísly reálnými je vždy třeba dávat pozor ...případné zaokrouhlovací chyby.

Způsob vnitřního uložení čísel závisí na konkrétním typu počítače ...použitém překladači, obecné zásady pro ukládání jsou ale vždy stejné. ...esný tvar reprezentace čísel v počítači se liší jednak velikostí paměťové...prostoru, který se pro jedno číslo vyhradí, jednak tím, jakým způsobem ...tento prostor využit.

Kladná celá čísla jsou zobrazena v paměti počítače ve dvojkové ...ustavě, záporná celá čísla se ukládají obvykle v doplňkovém kódu. ...ejvyšší bit slouží k uložení znaménka — je-li roven 0, je číslo kladné, je-li ...ven 1, číslo je záporné. Vlastní hodnota čísla je zobrazena ve zbývajících ...tech. Z nutného omezení velikosti paměti přidělované pro jedno číslo ...yne, že celočíselný datový typ v programech neslouží k ukládání „celých

čísel", jak se často zjednodušeně říká, ale pouze k ukládání „celých čísel z jistého rozmezí". Například jazyk Turbo Pascal používá pro uložení hodnot typu `integer` paměť velikosti 16 bitů. Z těchto šestnácti bitů je jeden bit znaménkový a ve zbývajících patnácti je zobrazena vlastní hodnota. V takto velkém paměťovém prostoru lze zobrazit libovolné celé číslo z rozmezí od $-2^{15}$ do $2^{15}-1$, tj. od $-32\,768$ do $32\,767$. Pokud by vám toto omezení rozsahu přípustných hodnot z jakýchkoli důvodů nevyhovovalo, můžete buď použít některý jiný z celočíselných datových typů (jazyk Turbo Pascal nabízí ještě čtyři další: `shortint`, `longint`, `byte` a `word`), nebo si musíte pomoci sami programově. Snadno si můžete naprogramovat prostředky pro aritmetické výpočty s vícecifernými čísly tak, že každé číslo uložíte po částech (po skupinách cifer) do jednotlivých složek pole nebo do prvků lineárního spojového seznamu.

Datový typ určený pro uložení „reálných čísel" nám přináší ještě více omezení. Ve skutečnosti slouží pouze pro práci s racionálními čísly, a to pouze s čísly zobrazenými na jistý předem omezený počet platných cifer a z jistého omezeného rozsahu přípustných hodnot. Výstižnější označení tohoto datového typu je „typ pro zobrazení čísel v pohyblivé řádové čárce". Čísla se ukládají do paměti počítače v podobě znaménka, mantisy a exponentu. Z celkového paměťového prostoru určeného pro uložení jednoho čísla slouží jeden bit na znaménko, pevně stanovený počet bitů je určen pro zobrazení exponentu a do zbytku místa se ukládá mantisa. Hodnota zobrazeného čísla se pak určí jako součin mantisy a základu umocněného na exponent a doplní se správným znaménkem. Jako základ se používá nejčastěji číslo 2. Mantisa obsahuje číslo v tzv. normalizovaném tvaru. Předpokládá se, že desetinná tečka je umístěna před nejvyšším řádem mantisy, tento nejvyšší řád mantisy je přitom nenulový. Exponent udává, o kolik míst je třeba posunout řádovou čárku. Je-li hodnota exponentu kladná, je zobrazované číslo v absolutní hodnotě větší než 1 a desetinnou čárku je nutné posunout doprava. Je-li hodnota exponentu záporná, je zobrazované číslo v absolutní hodnotě menší než jedna a desetinnou čárku posuneme doleva.

Není naším cílem věnovat se detailněji otázkám vnitřního zobrazení čísel. Důležité pro nás ale je vědět, že omezený počet bitů sloužících pro zobrazení mantisy určuje maximální počet platných cifer, na něž se číslo zobrazuje, a omezený počet bitů určených pro exponent vymezuje rozsah hodnot, které lze vůbec zobrazit. Minimální velikost paměti určené pro



uložení jednoho čísla v pohyblivé řádové čárce bývá na počítačích 4 byty. Z toho obvykle 24 bitů slouží pro uložení mantisy (včetně znaménka) a 8 bitů zabírá exponent. Z rozsahu mantisy lze zjistit přesnost zobrazení čísla. Je-li mantisa uložena ve 23 bitech, mohou se dvě sousední čísla lišit v hodnotě nejnižšího bitu, jehož váha je $2^{-22}$krát menší než váha nejvýznamnějšího bitu. V desítkovém vyjádření to odpovídá přibližně $2{,}4 \cdot 10^{-7}$. Z toho lze usoudit, že při tomto zobrazení lze používat čísla s přesností na 6 až 7 platných číslic. Rozsah velikosti čísel vyplývá z velikosti paměti určené pro uložení exponentu. Exponent zobrazený v 8 bitech představuje posun desetinné tečky v rozmezí od $-2^7$ do $2^7-1$, tj. $-128$ až 127. Zobrazitelná čísla tedy mohou ležet v rozmezí od $2^{-128}$ do $2^{127}$, což je přibližně $10^{-38}$ až $10^{38}$. Turbo Pascal oproti tomu vyhradí pro uložení jednoho čísla typu `real` 6 bytů (tj. 48 bitů). Z nich 40 bitů používá pro uložení mantisy a dosahuje tak vyšší přesnosti, přibližně 11 platných cifer. Na exponent zbývá opět 8 bitů, takže můžete pracovat s hodnotami z rozsahu přibližně $10^{-38}$ až $10^{38}$.

Jestliže provádíme nějaké výpočty s celými čísly, je dobré tento číselný obor neopouštět, pokud to není nezbytné. Jakýkoli dočasný přechod do oboru reálných čísel jednak znamená zpomalení výpočtu, jednak hrozí nebezpečí vzniku zaokrouhlovacích chyb. Pozor musíme dát zejména při dělení. Programovací jazyky nám obvykle dávají prostředky na celočíselné dělení, ale mnoho začínajících programátorů na ně s oblibou zapomíná. V následujících dvou příkladech si nyní ukážeme několik základních konstrukcí zapsaných v jazyce Pascal jednak správně, jednak nešikovně.

**Příklad 1**

Máme deklarovány proměnné $K$, $L$, $M$ typu `integer`. Proměnné $K$, $L$ mají přiřazenu kladnou hodnotu. Potřebujeme určit celou část podílu $K/L$. Správný postup využívá celočíselného dělení. V Pascalu stačí napsat

```
M:=K div L;
```

Mnoho začátečníků ovšem píše zbytečně složitě

```
M:=trunc(K/L);
```

**Příklad 2**

Máme deklarovány proměnné $K$, $L$ typu `integer`. Obě proměnné $K$, $L$ mají přiřazenu kladnou hodnotu. Potřebujeme zjistit, zda je číslo $K$ dělitelné číslem $L$. Ve správném řešení použijeme operátor zbytku po celočíselném dělení:

```
if K mod L = 0 then {...}
```

V podstatě správný, ale zbytečně složitý je postup se zpětným násobením podílu dělitelem:

```
if (K div L) * L = K then {...}
```

Vůbec ovšem nepotřebujeme pracovat s výrazy typu real, jako je tomu v následujících dvou nešikovných řešeních:

```
if (K div L) = (K/L) then {...}
if round(K/L) * L = K then {...}
```

## 4.2 Zaokrouhlovací chyby

Při provádění numerických výpočtů s reálnými čísly si musíme stále dobře uvědomovat, jak jsou reálná čísla v počítači reprezentována. Tomu musíme přizpůsobovat všechny naše výpočty v reálné aritmetice. Jinak nám hrozí, že teoreticky správné algoritmy budou vlivem zaokrouhlovacích chyb dávat chybné výsledky. V této knize se vůbec nebudeme věnovat algoritmům numerické matematiky, jejichž součástí bývá i analýza možných zaokrouhlovacích chyb a metody jejich odstraňování. Přesto bude užitečné ukázat si alespoň ve stručnosti některá úskalí, na která můžeme při práci s reálnými čísly narazit.

Jak už víme, pro uložení jednoho čísla v pohyblivé řádové čárce je v paměti počítače vyhrazeno místo pevně dané velikosti. V takovémto pevně vymezeném prostoru je samozřejmě možné zobrazit jen konečně mnoho různých hodnot, zatímco počet všech reálných čísel je nekonečný. Je tedy zřejmé, že zdaleka ne každé reálné číslo bude zobrazitelné. Omezena je jednak přesnost zobrazení čísla, tj. počet jeho platných cifer, jednak velikost zobrazitelných čísel, tj. přípustný rozsah řádů jejich cifer. Reálná čísla jsou v programu nahrazena racionálními čísly s určitým pevně



zvoleným počtem platných cifer a z určitého stanoveného rozmezí. S čísly, jejichž absolutní hodnota přesahuje toto rozmezí, nelze vůbec pracovat. Ostatní reálná čísla jsou při uložení do paměti počítače zaokrouhlena a nahrazena vždy nejbližším zobrazitelným racionálním číslem (nejbližší hodnotou příslušného datového typu, např. `real` v Pascalu).

Zaokrouhlování čísel si nejlépe předvedeme v několika malých programových ukázkách. Všechny tyto programy jsou napsány v Pascalu, číselné hodnoty v nich použité jsou přizpůsobeny vlastnostem jazyka Turbo Pascal. Při práci v jiném programovacím jazyce nebo s jiným překladačem Pascalu se setkáte se stejnými problémy, v případě odlišného způsobu uložení „reálného" datového typu však konkrétní číselné hodnoty mohou vycházet jinak. Vyzkoušejte si sami tyto jednoduché programy na počítači a případně si vhodně upravte v nich použité číselné konstanty. Pro někoho budou možná výsledky následujících programů trochu překvapující.

Nejprve si ukážeme samotný fakt zaokrouhlení hodnoty přímo vložené do proměnné. Přesvědčte se sami, že následující program vypíše hodnotu proměnné $R$ rovnou 1:

```
program Zaokrouhleni;
var X, Y, R: real;
begin
  X := 1.0;
  Y := 0.000000000005;
  R := X-Y;
  writeln(X, Y, R)
      { výsledek:
      1.0000000000E+00 5.0000000000E-12 1.0000000000E+00   }
end.
```

Také naše druhá ukázka předvádí, jak se projevuje omezený počet platných cifer, pokud chceme například sčítat čísla řádově různé velikosti:

```
program RuznaCisla;
var R: real;
begin
  R := 1E13;
  if R = R+1 then writeln('R=R+1')
end.
```

Hodnoty $1E13$ a 1 se liší natolik, že v součtu $R + 1$ se přičítaná jednička vůbec neprojeví a program ohlásí rovnost $R = R + 1$.

Ze stejných důvodů nemusí platit při sčítání ani asociativní zákon, tj. vztah $A + (B + C) = (A + B) + C$. Předvedeme si to opět na malém příkladě:

```
program Asociativita;
var A, B, C: real;
begin
  A :=  1;
  B :=  1E15;
  C := -1E15;
  writeln( A+(B+C) );      { výsledek: 1 }
  writeln( (A+B)+C );      { výsledek: 0 }
end.
```

Zdůrazněme ještě jednou, že problém zaokrouhlovacích chyb vyplývá z omezeného počtu platných cifer čísla, nikoli ze samotné velikosti ukládaného čísla. Laická představa bývá v tomto směru dosti zkreslená. Často se objevuje názor, že počítač nedokáže rozlišit pouze čísla, jejichž rozdíl je v absolutní hodnotě velmi malý, menší než nějaká daná malá konstanta (běžná představa se pohybuje řádově něco kolem $10^{-10}$). Následující program ukazuje, jak je tento názor mylný:

```
program VelkaStejnaCisla;
var R, S: real;
begin
  R := 1.00000000001E20;
  S := 1.00000000000E20;
  writeln(R, S)
end.
```

V paměti počítače se obě čísla zobrazí stejným způsobem, takže program vypíše dvě stejné hodnoty `1.0000000000E20`, přestože se obě zadaná čísla liší o miliardu.

Na závěr si ukážeme ještě jeden poměrně známý příklad, který ilustruje nebezpečí zaokrouhlovacích chyb. V trochu jiné podobě ho naleznete také například v [5]. Budeme sčítat $N$ čísel, která mají hodnotu $1/N$, pro některá malá $N$. Podle našich znalostí z matematiky bychom mohli



právem očekávat, že výsledek takového součtu bude vždy roven jedné. Snadno se ale přesvědčíme, že tomu tak zdaleka nemusí být:

```
program Soucet_Ntin;
var N, I: integer;
    Ntina, Suma : real;
begin
  write('          hodnota 1/N');
  writeln('              součet N hodnot 1/N');
  for N:= 1 to 20 do
    begin
    Ntina := 1/N;
    Suma := 0;
    for I := 1 to N do
      Suma := Suma + Ntina;
    write('1/', N, ' =', Ntina, '    ', Suma);
    if Suma = 1 then writeln('  = 1')
                else writeln(' <> 1')
    end
end.
```

Program vypíše pro každé $N$ od 1 do 20 hodnotu $1/N$, součet $N$ hodnot $1/N$ a informaci, zda je tento součet roven přesně jedné. Výsledky jsou velmi závislé na použitém způsobu uložení reálných čísel. Například jenom v samotném Turbo Pascalu získáte odlišné výsledky při práci s proměnnými typu real v režimu `$N-,$E-` (tj. bez numerického koprocesoru a jeho emulace) a v režimu `$N+,$E+` (tj. s numerickým koprocesorem nebo jeho emulací) a také při použití jiných reálných typů (`single`, `double`, `extended`). Ukážeme si, jak budou vypadat výsledky alespoň v prvních dvou uvedených případech.

Při práci bez koprocesoru se všechny zobrazené součty zdají být rovny přesně jedné. Porovnání s konstantou 1 však odhalí, že tomu tak není. Od přesné jedničky se liší na některém dalším desetinném místě, se kterým se ještě počítá, ale které se již nezobrazuje:

```
          hodnota 1/N              součet N hodnot 1/N
1/1 = 1.0000000000E+00    1.0000000000E+00  = 1
1/2 = 5.0000000000E-01    1.0000000000E+00  = 1
1/3 = 3.3333333333E-01    1.0000000000E+00  = 1
1/4 = 2.5000000000E-01    1.0000000000E+00  = 1
```

```
1/5 = 2.0000000000E-01     1.0000000000E+00  = 1
1/6 = 1.6666666667E-01     1.0000000000E+00  = 1
1/7 = 1.4285714286E-01     1.0000000000E+00 <> 1
1/8 = 1.2500000000E-01     1.0000000000E+00  = 1
1/9 = 1.1111111111E-01     1.0000000000E+00  = 1
1/10 = 1.0000000000E-01    1.0000000000E+00 <> 1
1/11 = 9.0909090909E-02    1.0000000000E+00  = 1
1/12 = 8.3333333333E-02    1.0000000000E+00 <> 1
1/13 = 7.6923076923E-02    1.0000000000E+00 <> 1
1/14 = 7.1428571429E-02    1.0000000000E+00 <> 1
1/15 = 6.6666666667E-02    1.0000000000E+00  = 1
1/16 = 6.2500000000E-02    1.0000000000E+00  = 1
1/17 = 5.8823529412E-02    1.0000000000E+00  = 1
1/18 = 5.5555555556E-02    1.0000000000E+00  = 1
1/19 = 5.2631578947E-02    1.0000000000E+00 <> 1
1/20 = 5.0000000000E-02    1.0000000000E+00 <> 1
```

Pokud použijeme přesnější zobrazení čísel, odlišnosti součtu od jedničky se stanou viditelnými:

```
          hodnota 1/N                  součet N hodnot 1/N
1/1 = 1.00000000000000E+0000    1.00000000000000E+0000  = 1
1/2 = 5.00000000000000E-0001    1.00000000000000E+0000  = 1
1/3 = 3.33333333333485E-0001    1.00000000000000E+0000  = 1
1/4 = 2.50000000000000E-0001    1.00000000000000E+0000  = 1
1/5 = 2.00000000000045E-0001    1.00000000000000E+0000  = 1
1/6 = 1.66666666666742E-0001    1.00000000000000E+0000  = 1
1/7 = 1.42857142857110E-0001    9.99999999999091E-0001 <> 1
1/8 = 1.25000000000000E-0001    1.00000000000000E+0000  = 1
1/9 = 1.11111111111086E-0001    1.00000000000000E+0000  = 1
1/10 = 1.00000000000023E-0001    1.00000000000182E+0000 <> 1
1/11 = 9.09090909091219E-0002    1.00000000000000E+0000  = 1
1/12 = 8.33333333333712E-0002    9.99999999998181E-0001 <> 1
1/13 = 7.69230769230944E-0002    9.99999999999091E-0001 <> 1
1/14 = 7.14285714285552E-0002    9.99999999999091E-0001 <> 1
1/15 = 6.66666666667197E-0002    1.00000000000000E+0000  = 1
1/16 = 6.25000000000000E-0002    1.00000000000000E+0000  = 1
1/17 = 5.88235294117680E-0002    1.00000000000000E+0000  = 1
1/18 = 5.55555555555429E-0002    1.00000000000000E+0000  = 1
1/19 = 5.26315789473415E-0002    1.00000000000182E+0000 <> 1
1/20 = 5.00000000000114E-0002    1.00000000000182E+0000 <> 1
```



Při pohledu na získané výsledky by nám mohlo připadat podivné, že ačkoli číslo 1/3 má nekonečný desetinný rozvoj, a musí se proto zobrazit zaokrouhleně ve tvaru 0,333 33... na jistý pevně zvolený počet desetinných míst, součet tří takových čísel dává přesně 1 a ne 0,999 99... Na druhé straně jednoduchá hodnota 1/10 nevychází rovna přesně 0,1 a součet deseti takových hodnot se jedničce nerovná. Tento zdánlivý paradox vyplývá ze skutečnosti, že zatímco my jsme zvyklí počítat a zobrazovat čísla v desítkové soustavě, počítač používá soustavu o základu 2 (tzv. dvojkovou nebo také binární soustavu). Při vyjádření v této soustavě mají nekonečný desetinný rozvoj jiná čísla než v soustavě desítkové, takže k problémům se zaokrouhlovacími chybami a k nepřesnostem při výpočtech dochází také u odlišných čísel než v desítkové soustavě.

### 4.3 Ordinální datové typy

Některé programovací jazyky umožňují pracovat vedle celočíselného datového typu ještě s dalšími tzv. ordinálními datovými typy. To jsou takové typy, které připouštějí pevně danou konečnou množinu hodnot s přirozeným očíslováním všech těchto hodnot celými čísly. Těmto číslům se potom říká ordinální (tj. pořadová) čísla jednotlivých hodnot. Ordinální čísla definují pořadí mezi hodnotami příslušného typu. Například v jazyce Pascal existují standardní ordinální typy `integer` pro uložení celých čísel ze stanoveného rozmezí, `char` pro zobrazení znaků a `boolean` pro uložení logické hodnoty `true/false`. Kromě toho si může každý definovat své vlastní výčtové typy, které jsou také ordinální.

Z hlediska vnitřního uložení představuje každý ordinální typ použitý v programu vlastně další celočíselný typ. Jeho hodnoty, zapisované v programovacím jazyce pro názornost symbolicky, jsou při překladu nahrazeny odpovídajícími ordinálními čísly.

## CVIČENÍ

**1.** Napište program, který bude počítat součet dvou velkých celých čísel. Můžete předpokládat, že sčítanci ani součet nebudou mít více než sto cifer. Návod: Dlouhá čísla ukládejte do pole. Každý prvek pole bude představovat jednu cifru nebo raději skupinu sousedních cifer. Při sčítání dejte pozor na správný výpočet přenosů do vyšších řádů.

**2.** Vyzkoušejte si sečíst $N$ čísel hodnoty $1/N$ a porovnejte výsledný součet s celočíselnou konstantou 1. Pokusy proveďte pro různě velká $N$ a pro různé druhy reálných číselných proměnných (v jazyce Turbo Pascal typy `real`, `single`, `double`, `extended`, to vše s numerickým koprocesorem, s jeho emulací nebo bez něj).



# 5 ZÁKLADNÍ DATOVÉ STRUKTURY

Pro ukládání hodnot používáme v programech vedle jednoduchých proměnných různé datové struktury. V této kapitole se přehledně seznámíme s nejdůležitějšími strukturami dat z pohledu jejich vnitřní realizace a ukážeme si základní algoritmické postupy, jak se s nimi pracuje. V dalších kapitolách knihy pak budeme všechny tyto struktury běžně používat v programech. V kap. 6 a kap. 7 navíc uvidíte jiný pohled na to, jak lze datové struktury rozlišovat — nikoli podle způsobu uložení dat v paměti, ale podle logiky přístupu k jednotlivým položkám dat (množina, zásobník, fronta).

Vzhledem k tomu, že tato kniha není a nechce být učebnicí žádného konkrétního programovacího jazyka, nebudeme se věnovat technickým detailům, jak se který datový typ v programu přesně deklaruje a používá. Ukážeme si pouze to podstatné, tj. co nám která datová struktura při programování umožňuje, co nového nám přináší a jak se s ní provádějí základní akce.

## 5.1 Pole, záznam, množina, řetězec

Tradičními datovými strukturami, které nám nabízí většina dnešních programovacích jazyků, jsou pole a záznam. K dalším známým strukturovaným datovým typům patří množina a řetězec.

**Pole** je nejdůležitější datová struktura, která se objevuje snad v každém netriviálním programu. Jde o skupinu položek stejného typu, které jsou pod společným názvem seřazeny v paměti za sebou jedna za druhou. Položky pole nemají svá vlastní pojmenování, odkazujeme se na ně pomocí indexů připojených ke jménu pole. Díky tomu, že všechny položky jsou stejného typu a zabírají tudíž stejné místo v paměti, je snadné určit přesné umístění zvolené položky pole v paměti na základě znalosti indexu této položky v rámci pole a adresy v paměti, na které je umístěn začátek pole.

Způsob indexování polí se v různých programovacích jazycích liší. Někdy se indexy počítají povinně od nuly nebo od jedničky, v jiných jazycích (např. také v Pascalu) je možné stanovit zvlášť pro každé pole hodnotu dolní i horní meze každého indexu. Některé jazyky používají pro indexování výhradně celá čísla, jiné nabízejí i širší možnosti (v Pascalu

můžete pro indexování použít libovolný ordinální typ). Programovací jazyk může, ale nemusí omezovat maximální možný počet indexů (Pascal neomezuje). V každém případě je ale indexování nová kvalita, která podstatným způsobem obohacuje vyjadřovací prostředky programovacího jazyka.

Pole používáme v programech všude tam, kde potřebujeme uchovávat větší množství údajů téhož typu a tyto údaje pak procházet, zpracovávat, vyhledávat v nich apod. V kap. 5.2 se podrobněji podíváme na různé možnosti algoritmického řešení vyhledávání v poli. Není ovšem vhodné použít pole o několika málo složkách místo odpovídajícího počtu jednoduchých proměnných, jestliže tyto proměnné mají společný pouze typ, ale logicky spolu nesouvisejí a nikde v programu je nepotřebujeme společným způsobem zpracovávat. V takovém případě bychom stejně nikdy nevyužili výhod, které nám pole dávají, a použitím indexované proměnné místo jednoduché bychom jenom zbytečně zkomplikovali a zpomalili výpočet.

Použití polí při řešení úloh na počítači je velmi rozmanité. Do pole reálných čísel si budeme ukládat třeba naměřené hodnoty, které má náš program vyhodnocovat a zpracovávat, v poli celých čísel si budeme uchovávat počty dní v jednotlivých měsících, jestliže bude program nějak přepočítávat kalendářní data. Pole znaků nám poslouží pro uložení jednoduchého obrázku či grafu, který potřebujeme opakovaně zobrazovat, nebo jako pomocná paměť pro ukládání zpracovávaného textu, když si budeme chtít naprogramovat jednoduchý textový editor. V poli znakových řetězců můžeme mít uložen například seznam jmen měsíců v roce, pokud potřebujeme opakovaně tisknout kalendářní data ve slovním vyjádření. Také k seřazení jakýchkoli zadaných hodnot podle velikosti použijeme pole. A až si budete chtít odpočinout od práce a naprogramovat si pro zábavu nějakou deskovou hru, jako je dáma nebo piškvorky, bez pole se také neobejdete — poslouží vám pro ukládání situace na hracím plánu.

Programovací jazyk může poskytovat prostředky buď pouze pro statická, nebo i pro tzv. **dynamická pole**. Dynamická pole známe třeba z Basicu. Slovo „dynamická" v jejich názvu znamená, že velikost těchto polí není známa při překladu, ale je určena až dynamicky během výpočtu. Počet položek pole tak může záviset například na některém vstupním údaji. V případě **statických polí** je nutné uvést v deklaraci pole jeho přesnou velikost. To znamená, že počet položek pole je konstantní a nemůže



záviset na údajích přečtených ze vstupu nebo určených při předchozích výpočtech. Programovací jazyk Pascal umožňuje vytvářet pouze takováto statická pole. Statická pole jsou výhodná z hlediska jejich vnitřní realizace, usnadňují paměťovou alokaci, a tím zrychlují výpočet, někdy však trochu omezují programátora. Pokud v programu potřebujeme vyhradit prostor pro uložení a další zpracování předem neznámého počtu údajů, nezbývá nám než nějakým způsobem odhadnout jejich maximální možný počet a deklarovat pak pole této velikosti. Jinou možností, jak tuto situaci řešit, je použít místo pole vhodnou dynamicky alokovanou datovou strukturu (spojový seznam — viz kap. 5.3). Slova „dynamicky alokovanou" zde opět naznačují, že konkrétní paměťový prostor je této struktuře přidělován až v průběhu výpočtu. Jeho velikost proto nemusí být předem pevně určena a může být stanovena až v době výpočtu. Dynamická pole užívaná např. v Basicu a dynamicky alokované datové struktury známé např. z programovacích jazyků Pascal a C mají společnou právě jen tuto „dynamičnost" v určení velikosti. Jinak jde ale o dva naprosto odlišné mechanismy hospodaření s pamětí počítače. My se budeme dále v této knize zabývat pouze druhým z nich.

**Záznam** je vlastně skupina různých datových položek, které patří logicky k sobě. Je proto vhodné udržovat je v programu pohromadě a mít možnost manipulovat s nimi v celku. Na rozdíl od pole mohou mít jednotlivé položky záznamu různé typy. Každá položka má své jméno a pomocí tohoto jména se na ni odkazujeme. Se záznamem můžeme pracovat jako s jedním celkem nebo zvlášť s každou jeho položkou.

Podobně jako pole i záznamy používáme v programech v mnoha různých situacích. Pomocí záznamu si můžete realizovat vlastní datový typ pro práci se zlomky (má celočíselné položky čitatel a jmenovatel) nebo s komplexními čísly (s položkami pro uložení reálné a imaginární části komplexního čísla). Kalendářní datum je typickým záznamem o třech celočíselných položkách (den, měsíc, rok). Naprosto klasickým záznamem je datová struktura sloužící k evidenci pracovníků — o každém se ukládá řada informací různých typů (např. jméno, datum narození, vzdělání, počet dětí, pracovní zařazení, plat apod.).

Jestliže v programu nepracujeme s dynamicky alokovanými proměnnými, nepřináší záznam z hlediska vyjadřovací síly jazyka vlastně nic nového a je spíše „kosmetickou" záležitostí. Pokud bychom z programovacího jazyka tuto konstrukci zcela vypustili, dokázali bychom všechny

programy celkem snadno upravit tak, aby místo se záznamem pracovaly s jeho položkami jako se samostatnými proměnnými. Pro přehlednost a srozumitelnost programu ovšem bývá použití záznamů velmi užitečné. Teprve ve spojení s dynamickou alokací proměnných však přednosti záznamů plně doceníme.

Poněkud méně obvyklou datovou strukturou je **množina**. V programech se vyskytuje dosti zřídka a mnohé programovací jazyky ji vůbec nepodporují. V jazyce Pascal ji ovšem máme k dispozici. Proměnná typu množina představuje skutečně množinu hodnot z předem známého základního (tzv. bázového) typu. Položky množiny musí být nějakého ordinálního typu a konkrétní implementace jazyka obvykle omezuje přípustný rozsah jejich ordinálních čísel. Například jazyk Turbo Pascal umožňuje vytvářet množiny z ordinálních hodnot s čísly od 0 do 256. Programovací jazyk nám pak nabízí nástroje pro přidání prvku do množiny, ubrání prvku, test náležení do množiny, množinové operace sjednocení, průnik a rozdíl množin. Z běžně používaných operací s množinami nám v Pascalu chybí pouze cyklus přes všechny prvky množiny. Ten jsme nuceni nahradit cyklem přes všechny hodnoty bázového typu spojený s testem náležení do množiny.

Výhodou datového typu množina je velmi úsporné uložení v paměti. Množina se ukládá ve tvaru bitové mapy. Každému prvku bázového typu odpovídá jeden bit v přidělené paměti. Jeho hodnota určuje, zda tento prvek do množiny patří, nebo ne. Také předem připravený aparát pro provádění základních množinových operací bývá velmi efektivní a využívá zpravidla bitových operací strojového kódu počítače.

Velmi často potřebujeme v programech pracovat se znakovými řetězci. **Řetězec** je posloupnost znaků tvořící jeden celek — třeba nějaké jméno, název nebo řádek zpracovávaného textu. S řetězcem pracujeme někdy jako s celkem, někdy po jednotlivých znacích, jimiž je tvořen.

Z hlediska vnitřní struktury i způsobu práce s ním připomíná řetězec jednorozměrné pole znaků. V normě jazyka Pascal se s proměnnými typu řetězec vůbec nepočítá (existují tam pouze řetězcové konstanty) a jsou nouzově nahrazovány právě znakovými poli. O trochu větší možnosti než obyčejné pole nám pro práci se znakovými řetězci dává tzv. **zhuštěné pole znaků**, tj. pole typu `packed array`. Tento typ bychom mohli charakterizovat nejlépe jako znakový řetězec pevně zvolené délky. Některé



překladače Pascalu však se zhuštěnými poli nepracují a slovo `packed` v deklaraci pole jednoduše ignorují.

Turbo Pascal odstranil nedostatky jazyka Pascal v práci se znakovými řetězci a zavedl samostatný plnohodnotný typ řetězec. I ten některými svými rysy připomíná pole znaků. Ke složkám řetězce, tj. k jeho jednotlivým znakům, přistupujeme stejným způsobem jako k položkám pole — indexováním. Navíc ale máme k dispozici bohatý aparát předdefinovaných operací, procedur a funkcí, pomocí nichž můžeme například velmi snadno řetězce spojovat nebo v řetězcích vyhledávat, vypouštět a vkládat podřetězce. Do proměnných typu řetězec můžeme číst hodnoty ze vstupních souborů a řetězce můžeme samozřejmě také zapisovat do výstupních souborů.

## 5.2 Vyhledávání v poli

Jednou z nejčastějších operací s poli je vyhledávání dané hodnoty. Předpokládejme, že máme pole celých čísel $M$ indexované od 1 do $N$ a máme zjistit, zda je v poli obsažena daná hodnota $X$. Pokud ano, máme určit index jejího výskytu (např. prvního). Ukážeme si nyní několik možných různě šikovných postupů. V celé kap. 5.2 budeme předpokládat platnost deklarací:

```
const N = 100;              {počet prvků v poli}
type  Pole = array[1..N] of integer;
```

1. For-cyklus vede k nejjednoduššímu zápisu programu, trochu nešikovné a neefektivní je ale další procházení polem $M$ i po nalezení hodnoty $X$:

```
function Vyhledani1(M: Pole; X: integer): integer;
  {v poli M[1..N] hledá hodnotu X}
  {pokud je X obsaženo v M, vrací její index v M}
  {pokud X není v M obsaženo, vrací nulu}
  var i,j: integer;
  begin
    j := 0;
    for i:=1 to N do
      if (M[i] = X) and (j = 0) then j:=i;
```

```
  Vyhledani1 := j;
end;
```

2. While-cyklus umožní ukončit průchod polem $M$ ihned po nalezení hodnoty $X$:

```
function Vyhledani2(M: Pole; X: integer): integer;
  {v poli M[1..N] hledá hodnotu X}
  {pokud je X obsaženo v M, vrací její index v M}
  {pokud X není v M obsaženo, vrací nulu}
  var i: integer;
  begin
    i := 1;
    while (i < N) and (M[i] <> X) do i:=i+1;
    if M[i] = X then Vyhledani2 := i {je obsaženo, X = M[i]}
               else Vyhledani2 := 0 {není obsaženo}
  end;
```

3. Pozor na while-cyklus v následující podobě:

```
function Vyhledani3(M: Pole; X: integer): integer;
  {v poli M[1..N] hledá hodnotu X}
  {pokud je X obsaženo v M, vrací její index v M}
  {pokud X není v M obsaženo, vrací nulu}
  var i: integer;
  begin
    i := 1;
    while (i <= N) and (M[i] <> X) do i:=i+1;
    if i <= N then Vyhledani3 := i {je obsaženo, X = M[i]}
              else Vyhledani3 := 0 {není obsaženo}
  end;
```

Funkce *Vyhledani3* vypadá zdánlivě správně, a dokonce lépe než předchozí funkce *Vyhledani2*, závěrečný test je v ní jednodušší. Správná však není a v některých případech může způsobit běhovou chybu při výpočtu! Pokud není v poli $M$ hodnota $X$ obsažena, dojde při běhu programu k přetečení horní meze pole. Tato chyba nastane při tzv. úplném vyhodnocování logických výrazů, které předpokládá norma jazyka Pascal. Když proměnná $i$ nabyde hodnoty $N+1$, test $(i \leqq N)$ dává sice



výsledek `false`, ale ještě se vyhodnocuje druhá část podmínky ($M[i] \neq X$). Problémy tohoto typu jsou v jazyce Turbo Pascal odstraněny zavedením možnosti zvolit si tzv. zkrácené vyhodnocování logických výrazů (třeba pomocí přepínače `$B`). Výraz je pak vyhodnocován zleva doprava jenom tak dlouho, dokud není z hodnot dílčích podvýrazů zřejmá celková výsledná hodnota. Naše programová ukázka by při nastavení zkráceného vyhodnocování logických výrazů pracovala správně.

4. Zavedením pomocné logické proměnné pro řízení cyklu nám odpadnou problémy s vyhodnocováním složených podmínek:

```
function Vyhledani4(M: Pole; X: integer): integer;
  {v poli M[1..N] hledá hodnotu X}
  {pokud je X obsaženo v M, vrací její index v M}
  {pokud X není v M obsaženo, vrací nulu}
  var i: integer;
      Dalsi: boolean;
  begin
    i := 1;
    Dalsi := true;
    while Dalsi do
      if M[i] = X then
        begin
          Dalsi := false;
          Vyhledani4 := i {X je v poli obsaženo, X = M[i]}
        end
      else if i = N then
        begin
          Dalsi := false;
          Vyhledani4 := 0 {X není v poli obsaženo}
        end
      else
        i:=i+1;  {testovat další prvek}
  end;
```

5. Při vyhledávání pomocí **zarážky** dočasně vložíme do pole hned za uložená data hledanou hodnotu. Musí tam být na ni ovšem ponecháno místo (pole tedy musíme deklarovat o jeden prvek větší!). Prohledávání se nejpozději na místě této zarážky rozhodně zastaví. Test v cyklu bude proto jednodušší, neboť nemusíme kontrolovat překročení indexu. Hledání bude tudíž probíhat rychleji:

```
function Vyhledani5(M: Pole; X: integer): integer;
  {v poli o rozsahu M[1..N+1] se zkoumanými  hodnotami
   uloženými v úseku M[1..N] hledá hodnotu X}
  {pokud je X obsaženo v M, vrací její index v M}
  {pokud X není v M obsaženo, vrací nulu}
  var i: integer;
  begin
    M[N+1] := X;   {zarážka}
    i := 1;
    while M[i] <> X do i:=i+1;
    if i <= Pocet then Vyhledani5 := i {je obsaženo, X=M[i]}
                  else Vyhledani5 := 0 {není obsaženo}
  end;
```

Všechny uvedené postupy vyhledávání hodnoty v poli mají lineární časovou složitost. V nejméně příznivém případě, tj. pokud se hledaná hodnota v poli nenachází, musíme pokaždé projít postupně celým polem a provést při tom přibližně $N$ porovnání.

Pokud nechceme obsah pole příliš často měnit, ale zato v poli potřebujeme opakovaně vyhledávat různé hodnoty, vyplatí se nám uspořádat hodnoty uložené v poli podle velikosti. Pole uspořádáme pomocí některého z třídicích algoritmů, s nimiž se také později seznámíme (v kap. 11). K vyhledávání v uspořádaném poli používáme metodu tzv. **binárního vyhledávání** založenou na myšlence půlení intervalů. V každém kroku výpočtu se zmenší na polovinu rozmezí indexů v poli $M$, kde může hledaná hodnota $X$ ležet, pokud je v poli $M$ vůbec obsažena. Na začátku uvažujeme všech $N$ prvků pole, po prvním kroku $N/2$, po druhém $N/4$ atd., dokud se toto rozmezí nezúží na jediný prvek, který již snadno otestujeme. Počet takových kroků je roven horní celé části z $\log_2 N$ (srov. s kap. 2.3 — výška binárního stromu), takže metoda má velmi příznivou logaritmickou časovou složitost.

```
function BinVyhledavani(M: Pole; X: integer): integer;
  {ve vzestupně uspořádaném poli M[1..N] hledá hodnotu X}
  {pokud je X obsaženo v M, vrací její index v M}
  {pokud X není v M obsaženo, vrací nulu}
  var i,j,k: integer;
  begin
```



```
    i := 1;       {dolní mez}
    j := N;       {horní mez}
    repeat
      k := (i+j) div 2;   {index prostředního prvku}
      if X > M[k] then i := k+1
                  else j := k-1
    until (M[k] = X) or (i > j);
    if M[k] = X then BinVyhledavani := k
              else BinVyhledavani := 0
  end;
```

Při programování algoritmu binárního vyhledávání je nutné velmi pečlivě hlídat správný pohyb indexů a nastavení podmínek. V předchozí programové ukázce by se nám například mohl na první pohled zdát logičtější podmíněný příkaz ve tvaru

```
if X > M[k] then i := k+1
            else j := k
```

neboť když neplatí $X > M[k]$, je jistě $X \leqq M[k]$, a měli bychom tedy hledat hodnotu $X$ v intervalu od indexu $i$ až do $k$ včetně. Pokud bychom ale takto upravili náš program, přestal by správně fungovat! Kdybychom se pokusili hledat v uspořádaném poli $M$ hodnotu $X$ menší než $M[1]$, výpočet by se nezastavil a uvázl by ve věčném cyklu s hodnotami proměnných $i = j = k = 1$.

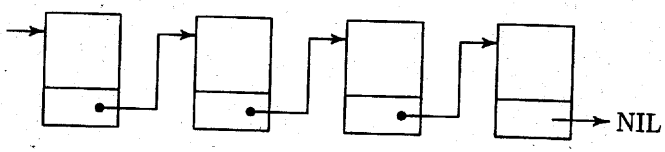
### 5.3 Lineární spojový seznam

Pomocí dynamicky alokovaných proměnných můžeme v programech vytvářet dynamické datové struktury. Proti klasickým strukturovaným datovým typům, s nimiž jsme se seznámili v kap. 5.1, mají dynamické struktury několik výhod i nevýhod. Výhodou je, že se potřebný paměťový prostor alokuje dynamicky až v době výpočtu, takže jeho velikost lze operativně přizpůsobit skutečným nárokům programu. Nástroje pro práci s dynamickými proměnnými navíc umožňují snadno vytvářet a obhospodařovat i složitější logické vazby a komplikovanější struktury (stromy, grafy apod.). Nevýhodou naopak je trochu více práce pro programátora při psaní programů, a zejména pak vyšší časové i paměťové nároky programu. Každé přidělení či uvolnění dynamicky alokované proměnné se

totiž provádí až během výpočtu a ve srovnání například s elementárními aritmetickými operacemi je dosti časově náročné. V dynamické datové struktuře se navíc používá část přiděleného paměťového prostoru na uložení ukazatelů, pomocí nichž je struktura vybudována a provázána. Paměťové náklady na „režii" mohou být v některých případech dosti vysoké.

Základní dynamickou datovou strukturou je **lineární spojový seznam**. Je to posloupnost záznamů stejného typu seřazených za sebe. V programech zastává často podobnou funkci jako jednorozměrné pole. Na rozdíl od pole nám ovšem neumožňuje přímý přístup k jednotlivým položkám pomocí indexů, spojovým seznamem musíme vždy procházet sekvenčně od začátku. Nejjednodušší podobou lineárního spojového seznamu je seznam **jednosměrný**, kdy do každého uzlu seznamu umístíme potřebnou ukládanou informaci a jeden ukazatel na následující uzel. Nesmíme zapomenout nastavit ukazatel v posledním uzlu seznamu na speciální hodnotu `nil`, která signalizuje konec seznamu.

```
type Uk = ^Uzel;
     Uzel = record
              Info: T;        {libovolný typ, co ukládáme}
              Dalsi: Uk       {následující uzel v seznamu}
            end;
```

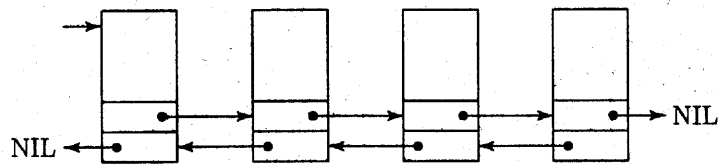



*Obr. 7* Lineární spojový seznam

Pro některé účely se hodí použít seznam **obousměrný**. V něm obsahuje každý uzel dva ukazatele — na svého předchůdce a na následníka v seznamu. Obousměrný seznam je paměťově náročnější, ale umožňuje zato procházet seznamem oběma směry a snáze se také provádějí některé další akce (např. vypuštění prvku).



```
type Uk = ^Uzel;
     Uzel = record
              Info: T;        {libovolný typ, co ukládáme}
              Pred: Uk;       {předcházející uzel v seznamu}
              Dalsi: Uk       {následující uzel v seznamu}
            end;
```




*Obr. 8* Obousměrný lineární spojový seznam

Ukážeme si nyní alespoň některé základní operace s jednosměrným lineárním spojovým seznamem. Pro jednoduchost budeme předpokládat, že položky *Info* v jednotlivých uzlech jsou typu `integer`.

1. Vytvoření lineárního spojového seznamu s $N$ prvky. Seznam bude obsahovat v položkách *Info* rostoucí hodnoty od 1 do $N$. Nejjednodušší je vytvářet seznam od konce, nové prvky přidáváme na začátek dosavadního seznamu.

```
procedure Vytvor1(N: integer; var P: Uk);
  {P bude ukazovat na začátek vytvořeného seznamu}
var Q: Uk;
begin
P := nil;              {na začátku je seznam prázdný}
while N > 0 do
  begin
  new(Q);
  Q^.Info := N;
  Q^.Dalsi := P;       {přidáme prvek Q na začátek seznamu P}
  P := Q;
  N := N-1
  end                  {P ukazuje na vytvořený seznam}
end;
```

2. Vytvoření lineárního spojového seznamu s $N$ prvky ve směru od začátku ke konci. Je to stejná akce jako v bodě 1, prvky vytvořeného seznamu budou opět obsahovat postupně celá čísla od 1 do $N$. Seznam budeme tentokrát ale vytvářet přidáváním nových prvků na konec seznamu.

```
procedure Vytvor2(N: integer; var P: Uk);
  {P bude ukazovat na začátek vytvořeného seznamu}
var Q,R:Uk;
    I:integer;
   {P - začátek seznamu (první prvek)
    Q - konec seznamu (poslední prvek)
    R - nový prvek                          }
begin
new(P);              {první prvek seznamu vytvoříme odděleně}
P^.Info := 1;
P^.Dalsi := nil;
Q := P;              {první prvek je zatím zároveň posledním}
for I:=2 to N do
  begin
  new(R);
  R^.Info := I;
  Q^.Dalsi := R;
  Q := R
  end;
Q^.Dalsi := nil   {ještě řádně zakončit seznam}
end;
```

3. Vkládání prvku do lineárního spojového seznamu za daný prvek. K vykonání této akce stačí provést dva přiřazovací příkazy (pozor na jejich pořadí!).

```
procedure VlozZa(P, Q: Uk);
  {P ukazuje na prvek seznamu, za který se má vkládat
   Q ukazuje na nově vkládaný prvek                   }
begin
Q^.Dalsi := P^.Dalsi;
P^.Dalsi := Q;
end;
```



4. Vkládání prvku do lineárního spojového seznamu před daný prvek je složitější, neboť nemůžeme v seznamu postupovat proti směru zřetězení prvků. Předpokládejme opět, že ukazatel $P$ ukazuje na prvek, před který chceme vložit nový prvek, a ukazatel $Q$ ukazuje na nově vkládaný prvek. Buď musíme projít celým seznamem od začátku, nalézt předchůdce prvku $P$ a za něj vložit $Q$ postupem uvedeným v bodě 3, nebo použijeme malý trik: Nový prvek zapojíme do seznamu až za prvek, na který ukazuje $P$, a potom vyměníme obsah položek Info v uzlech $P^\wedge$ a $Q^\wedge$.

```
procedure VlozPred(P, Q: Uk);
  {P ukazuje na prvek seznamu, před který se má vkládat
   Q ukazuje na nově vkládaný prvek                   }
var K: T;          {T je typ položky Info v uzlu seznamu}
begin
K := Q^.Info;      {do proměnné K si odložíme obsah Q^}
Q^ := P^;          {kopírují se celé uzly, tj. Info i Dalsi}
P^.Info := K;      {dokončit výměnu hodnot Info}
P^.Dalsi := Q;     {zapojit nový prvek}
end;
```

5. Vypuštění prvku z lineárního spojového seznamu je snadné, pokud známe předchůdce vypouštěného uzlu.

```
function Vypust(P: Uk): Uk;
  {P ukazuje na předchůdce rušeného prvku}
  {Funkce vrací ukazatel na vypuštěný prvek, příp. vrací
   hodnotu nil, pokud byl P posledním prvkem seznamu}
begin
Q := P^.Dalsi;                {Q ukazuje na rušený prvek}
if Q <> nil then
  begin
  P^.Dalsi := Q^.Dalsi;       {uzel Q je vypuštěn ze seznamu}
  Q^.Dalsi := nil;            {odpojit před dalším použitím}
  end;
Vypust := Q
end;
```

6. Vypuštění prvku z lineárního spojového seznamu je těžší, jestliže neznáme ukazatel na předchůdce vypouštěného prvku. Máme vypustit prvek, na který ukazuje ukazatel $P$. Podobně jako v bodě 4 se opět

nabízejí dvě možná řešení. Buď můžeme projít seznamem od začátku, nalézt předchůdce vypouštěného prvku $P^\wedge$ a dále postupovat stejně jako v bodě 5, nebo použijeme podobný trik jako v bodě 4: Hodnota následníka uzlu $P^\wedge$ se přesune do $P^\wedge$ a v seznamu se pak zruší následník uzlu $P^\wedge$. Tento trik ovšem nelze použít, chceme-li zrušit poslední prvek seznamu (nemá následníka).

7. Průchod seznamem. Máme za úkol projít seznamem a s položkou *Info* každého prvku provést jistou akci $X$.

```
procedure Pruchod(P: Uk);
  {P - začátek seznamu}
begin
while P <> nil do    {test na konec seznamu}
  begin
  X(P^.Info);        {požadovaná akce s prvkem seznamu}
  P:=P^.Dalsi
  end
end;
```

8. Hledání prvku s danou vlastností. V lineárním spojovém seznamu máme nalézt první prvek takový, jehož položka *Info* je rovna dané hodnotě $X$.

```
function Vyhledej1(P: Uk; X: integer): Uk;
  {P - začátek seznamu, X - hledaná hodnota}
  {vrací ukazatel na první prvek s Info hodnotou rovnou X}
  {pokud X není v seznamu, vrací nil}
var Q: Uk;
begin
Q := P;
while (Q <> nil) and (Q^.Info <> X) do Q := Q^.Dalsi;
Vyhledej1 := Q
end;
```

Funkce *Vyhledej1* není obecně správným řešením našeho úkolu. Není-li totiž hodnota $X$ v seznamu obsažena, může výpočet skončit běhovou chybou. Jestliže bude složená podmínka while-cyklu vyhodnocována metodou úplného vyhodnocení, jak to předpokládá norma jazyka Pascal, nabyde na konci seznamu proměnná $Q$ hodnoty `nil`, ale v tom okamžiku bude ještě chybně testováno, zda je $Q^\wedge$.*Info* různé od $X$. Uvedená funkce



je ovšem zcela v pořádku v jazyce Turbo Pascal, pokud je nastaveno zkrácené vyhodnocování logických výrazů (například přepínačem `$B-`). Je to obdobná situace, s jakou jsme se již setkali při vyhledávání v poli v kap. 5.2.

Ukážeme si ještě druhé řešení úlohy, které pracuje spolehlivě za všech okolností. Podobně jako v případě polí využívá pomocnou logickou proměnnou.

```
function Vyhledej2(P: Uk; X: integer): Uk;
  {P - začátek seznamu, X - hledaná hodnota}
  {vrací ukazatel na první prvek s Info hodnotou rovnou X}
  {pokud X není v seznamu, vrací nil}
var Q: Uk;
    Nenalezen: boolean;
begin
Q := P;
Nenalezen := true;
while (Q <> nil) and Nenalezen do
  if Q^.Info = X then Nenalezen := false
                 else Q := Q^.Dalsi;
Vyhledej2 := Q
end;
```

Také při práci s lineárními spojovými seznamy můžeme použít vyhledávání pomocí **zarážky**. Jde o stejný obrat, jaký jste viděli již v kap. 5.2 při vyhledávání v poli. Jestliže budeme chtít v nějakém seznamu opakovaně vyhledávat, připojíme si na jeho konec jeden pomocný prvek ve funkci zarážky. Odkaz na něj si budeme udržovat v další pomocné proměnné. Při každém novém vyhledávání vložíme nejprve hledanou hodnotu do zarážky a pak již snadno procházíme seznamem, aniž bychom se museli starat o konec seznamu. Máme totiž jistotu, že hledanou hodnotu v seznamu najdeme — když ne dříve, tak v přidané zarážce.

```
function Vyhledej3(P, R: Uk; X: integer): Uk;
  {P - začátek seznamu, R - zarážka, X - hledaná hodnota}
  {vrací ukazatel na první prvek s Info hodnotou rovnou X}
  {pokud X není v seznamu, vrací nil}
var Q: Uk;
begin
```

```
R^.Info := X;          {naplnění zarážky hledanou hodnotou}
Q := P;
while Q^.Info <> X do Q := Q^.Dalsi;
if Q = R then
  Vyhledej3 := nil     {našel hodnotu X až v zarážce}
else
  Vyhledej3 := Q       {hodnota X nalezena v seznamu}
end;
```

### 5.4 Dynamická reprezentace stromu a grafu

S pojmy strom a graf jste se mohli alespoň v hrubých rysech seznámit v kap. 2.2. V mnoha programech potřebujeme pracovat s datovými strukturami, které mají právě charakter stromů nebo obecných grafů. Stromy i grafy lze v programu reprezentovat různými způsoby, jak pomocí polí, tak i dynamicky. V kap. 9, která je celá věnována problematice grafů a grafových algoritmů, se blíže seznámíme s několika možnostmi, jak lze graf v programu uložit a k čemu je který způsob uložení výhodný. Nyní si ukážeme, jak může vypadat dynamická reprezentace stromů a grafů v programu.

Nejjednodušší strukturou, která nás bude nyní zajímat, je **binární strom**. Každý uzel stromu obsahuje vedle vlastní uložené informace ještě dva ukazatele — na svého levého a pravého syna. Pokud uzel nemá některého ze synů, příslušný ukazatel dostane hodnotu `nil`. Speciálně v listech stromu jsou oba ukazatele nastaveny na `nil`.

```
type Uk = ^Uzel;
     Uzel = record
              Info: T;        {uložená informace}
              L,P : Uk        {levý a pravý následník}
            end;
```

S takto reprezentovanými binárními stromy se v této knize ještě několikrát setkáte. Hned v kap. 6 použijeme tzv. binární vyhledávací stromy pro vnitřní uložení množiny údajů a předvedeme si, jak se s nimi pracuje. V kap. 8 si ukážeme prohledávání stromů do hloubky a do šířky, v kap. 12 uvidíte konkrétní praktickou aplikaci binárních stromů při reprezentaci aritmetických výrazů v programu.



V **obecných stromech** může mít jeden uzel více následníků. V programu pro ně můžeme použít třeba některou z následujících reprezentací:

1. Jestliže známe maximální možný počet $P$ následníků každého uzlu a $P$ je přitom malé, umístíme do uzlu pole ukazatelů na následníky.

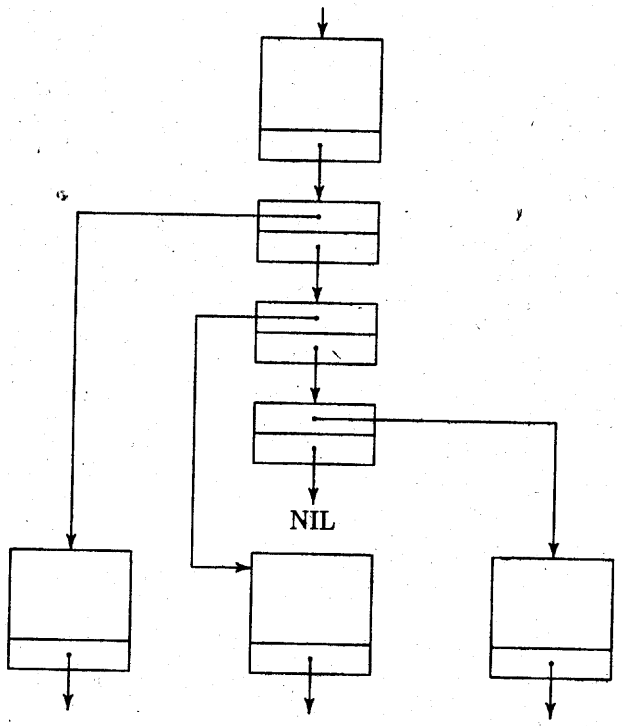
```
type Uk = ^Uzel;
     Uzel = record
              Info: T;          {uložená informace}
              Naslednik: array[1..P] of Uk;
            end;
```

2. V obecném případě nahradíme raději toto pole pomocným lineárním spojovým seznamem. V každém uzlu stromu bude uložen odkaz na začátek tohoto seznamu. Prvky seznamu pak obsahují ukazatele na jednotlivé následníky tohoto uzlu.

```
type Uk = ^Uzel;
     UkHr = ^Hrana;
     Uzel = record                {uzly stromu}
              Info: T;            {uložená informace}
              Hr: UkHr            {seznam následníků}
            end;
     Hrana = record               {prvky pomocného seznamu}
               Nasl: Uk;          {následník uzlu stromu}
               Dalsi: UkHr        {spojení prvků seznamu}
             end;
```

3. Tzv. **kanonická reprezentace** stromu nahrazuje jakýkoli obecný strom stromem binárním. Využívá k tomu zajímavý trik spočívající v tom, že v každém uzlu stromu je uložen odkaz na prvního syna a na „bratra" tohoto uzlu. Z libovolného uzlu se tedy dostaneme ke všem jeho následníkům tak, že přejdeme na prvního z nich po ukazateli *Syn* a mezi následníky pak přecházíme pomocí jejich ukazatelů *Bratr*.

```
type Uk = ^Uzel;
     Uzel = record
              Info: T;          {uložená informace}
              Syn, Bratr: Uk
            end;
```

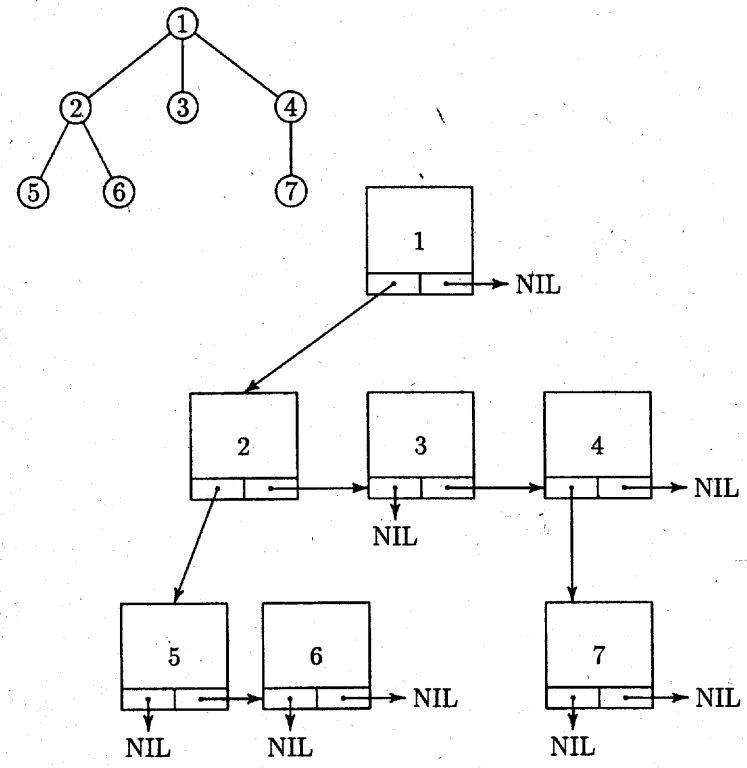



*Obr. 9* Reprezentace obecného stromu (uzel se třemi následníky)

Dynamická reprezentace **grafu** se z hlediska formální deklarace datových struktur vůbec neliší od reprezentace obecného stromu. Rozdíl je pouze na úrovni logické, jak jednotlivé exempláře typu *Uzel* pospojujeme. Zatímco u stromů měl každý uzel právě jednoho svého předchůdce (s výjimkou kořene stromu, který žádného předchůdce nemá), u grafu nemusí být tato stromová struktura dodržena.

Každý graf v dynamické reprezentaci je přirozeným způsobem graf orientovaný. Ukazatel směřující od uzlu grafu k jeho následníkovi přímo určuje orientaci této hrany. Jestliže chceme reprezentovat dynamicky graf neorientovaný, musíme ho nahradit příslušným orientovaným grafem. Místo neorientované hrany původního grafu vedoucí mezi vrcholy $A$ a $B$






*Obr. 10* Kanonická reprezentace stromu

bude mít nový orientovaný graf hrany dvě: jedna bude směřovat z vrcholu $A$ do vrcholu $B$, druhá naopak z $B$ do $A$.

Algoritmy pro práci se stromy a grafy nejsou již tak jednoduché, jak tomu bylo u lineárních spojových seznamů. Nebudeme je zde proto více rozebírat a ukážeme si je pokaždé až v situaci, kde je budeme potřebovat při řešení nějaké konkrétní úlohy (v kap. 6, 8, 9, 12).

## 5.5 Objekty

Významnou datovou strukturou moderních programovacích jazyků jsou tzv. **objekty**. Po formální stránce jsou objekty dosti podobné záznamům. Liší se od nich pouze tím, že vedle datových položek zahrnují navíc také zvláštní položky zvané metody. Metodou objektu může být procedura nebo funkce, která zpravidla nějak manipuluje s objektem a s jeho datovými položkami. Do jednoho záznamu je teoreticky možné zařadit položky, které spolu nijak logicky nesouvisejí. Takové použití záznamů ovšem nemá rozumný smysl a pro srozumitelnost a přehlednost programu je jen škodlivé. Stejně tak metodami objektu mohou být teoreticky vzato zcela libovolné procedury nebo funkce. Smysluplné použití objektů ovšem spočívá v tom, že do jednoho objektu zahrneme jako metody ty procedury a funkce, které se bezprostředně vztahují k datovým položkám tohoto objektu. Jeden celek tak vytvoří data a procedury, které patří logicky k sobě.

Zavedení objektů do programovacích jazyků přineslo do programování nové vyjadřovací možnosti a kvality. Znamenalo vznik nového přístupu k samotnému návrhu a tvorbě programů. Tato metodika využívající všech nových vlastností objektů a idejí abstraktních datových typů se nazývá **objektové programování**. Podrobný výklad myšlenek objektů a objektového programování by vydal na samostatnou knihu, proto se mu zde nebudeme více věnovat. Rovněž v programových ukázkách uvedených v této knize nebudeme objekty používat. Norma jazyka Pascal objekty vůbec neobsahuje, jsou ale jedním z mnoha rozšíření jazyka Turbo Pascal proti normě.

Pro úplnost ještě dodejme, že slovo objekt se i v programování běžně užívá také ve zcela obecném významu, jak jsme na to zvyklí z jiných oblastí. Ze souvislostí je pak nutné rozlišit, kdy se hovoří o zmíněné speciální datové struktuře a kdy ne.

## CVIČENÍ

**1.** Definujte si datový typ pro uložení komplexních čísel a napište procedury pro sčítání a násobení komplexních čísel. *Návod*: Použijte záznam o dvou položkách.

**2.** Napište program pro určování, kolik dní uplyne mezi danými dvěma daty. Uvažujte počty dní v jednotlivých měsících a počítejte také



s přestupnými roky. Od roku 1582 používáme gregoriánský kalendář, podle kterého je přestupný každý rok, jehož číslo je dělitelné čtyřmi, s výjimkou celých století, která přestupná nejsou. Ale i zde je ještě jedna výjimka: roky dělitelné beze zbytku číslem 400 přestupné jsou, přestože to jsou také celá století.

**3.** Napište proceduru pro vyhledání dané hodnoty $X$ ve vzestupně uspořádaném poli celých čísel. Použijte při tom sekvenční procházení polem. *Návod*: Skutečnost, že je pole uspořádané, využijete k předčasnému ukončení průchodu polem ve chvíli, kdy hodnota prvku v poli překročí hledané $X$.

**4.** Napište proceduru, která obrátí pořadí prvků v daném jednosměrném lineárním spojovém seznamu celých čísel. Ze seznamu obsahujícího po řadě hodnoty 2, 8, 4, 9 tedy vytvoří seznam s prvky v pořadí 9, 4, 8, 2. *Návod*: Původní seznam rozebírejte od začátku prvek po prvku a z odebíraných prvků původního seznamu ihned vytvářejte odzadu výsledný seznam.

**5.** Napište proceduru, která korektně zruší zadaný již nepotřebný lineární spojový seznam. *Návod*: Seznam je třeba rušit postupně prvek po prvku, aby v paměti počítače nezůstala obsazena žádná z programu nedostupná paměťová místa.

**6.** Napište proceduru, která v jednosměrném lineárním spojovém seznamu celých čísel vyhledá první uzel s hodnotou položky *Info* rovnou zadanému $X$ a tento uzel zruší. Pokud nebude hodnota $X$ v seznamu vůbec obsažena, procedura ponechá původní seznam beze změn. *Návod*: Procházejte zadaným seznamem od začátku a hledejte v něm hodnotu $X$. Průběžně si při tom udržujte odkaz na předchůdce zkoumaného uzlu, abyste po nalezení $X$ dokázali uzel s hodnotou $X$ snadno vypustit.

**7.** Napište proceduru, která v jednosměrném lineárním spojovém seznamu celých čísel vyhledá všechny uzly s hodnotou položky *Info* rovnou zadanému $X$ a tyto uzly zruší. Pokud nebude hodnota $X$ v seznamu vůbec obsažena, procedura ponechá původní seznam beze změn.

# 6 UKLÁDÁNÍ A VYHLEDÁVÁNÍ ÚDAJŮ

Při návrhu algoritmu a psaní programu je důležitá správná volba datových struktur. Vhodný způsob uložení vstupních dat i mezivýsledků může zásadním způsobem ovlivnit rychlost výpočtu. Při výběru datových struktur je třeba vycházet z toho, jaké operace budeme s daty provádět. Potřeby jednotlivých prováděných operací jsou často protichůdné, takže výsledná volba datové struktury bývá vhodně zvoleným kompromisem.

V této kapitole se budeme věnovat nejzákladnější programátorské technice — metodám ukládání dat a jejich vyhledávání. Budeme řešit následující úkol, se kterým se opakovaně setkáváme v mnoha programech. Potřebujeme vhodně reprezentovat jistou množinu údajů tak, abychom s ní mohli provádět řadu operací: přidat do ní další prvek, vynechat z ní daný prvek a samozřejmě také otestovat, zda je v množině jistá hodnota uložena, a pokud ano, vyhledat ji. Další operací s množinou dat může být průchod přes všechny prvky množiny spojený s vykonáním nějaké akce s každým z prvků.

Ukládaná data nemusí vždy tvořit množinu. Vedle množin používáme pro ukládání dat také seznamy. Na rozdíl od množin v seznamech záleží na pořadí jednotlivých prvků a připouští se navíc opakovaný výskyt téže hodnoty. Tato kapitola bude pojednávat o možných způsobech reprezentace množin údajů, seznamům bude věnována následující kapitola.

Pro jednoduchost budeme nadále předpokládat, že chceme ukládat a vyhledávat celá čísla. Tytéž postupy se samozřejmě uplatní i v případě, že ukládané informace jsou jiného typu. V praxi bude často ukládanou informací záznam obsahující více položek. Některá z těchto položek nám poslouží jako **klíč**, podle něhož budeme údaje vyhledávat (tzn. budeme hledat ten záznam, jehož klíčová položka má jistou danou hodnotu).

Nabízí se řada možností, jak lze množinu a příslušné operace s ní realizovat. Rozebereme si zde alespoň ty nejzákladnější a nejpoužívanější. Pokaždé uvedeme případná omezení použitelnosti a posoudíme časovou složitost jednotlivých operací. Podobný, avšak více teoreticky zaměřený rozbor lze nalézt také v [13].



## 6.1 Pole příznaků

Množinu můžeme reprezentovat pomocí příznaků přímo určujících, který prvek je v množině obsažen. Tato reprezentace množiny je ze všech nejefektivnější, je však použitelná jen v jistých speciálních případech. Ukládanými informacemi (klíči) mohou být pouze celá čísla, popř. jiné jednoduché hodnoty, jako třeba znaky, a navíc ještě jen z předem známého ne příliš velkého rozmezí. Množinu takových čísel reprezentujeme vlastně polem logických hodnot. Pole je přímo indexováno ukládanými čísly z daného rozmezí. Uložené logické hodnoty udávají, který prvek je v množině obsažen. Znalost rozmezí povolených hodnot je nutná, aby bylo vůbec možné pole deklarovat. Rozmezí nesmí být příliš velké, aby pole nezabíralo příliš místa v paměti, resp. aby se do paměti vůbec vešlo.

Této reprezentaci se někdy říká také „charakteristická funkce množiny". Je použita i pro vnitřní reprezentaci standardního datového typu množina (`set`) v Pascalu. Výhodou metody je maximální možná časová i paměťová efektivita. Operace přidání prvku do množiny, vynechání prvku z množiny i testu náležení do množiny se provedou triviálně v konstantním čase přímým indexováním pole. Časově náročnější je průchod přes všechny prvky pole. Tento algoritmus je sice „lineární", počet elementárních operací ale není úměrný počtu prvků množiny, nýbrž celkovému rozsahu povolených hodnot. Musíme totiž projít celé pole reprezentující množinu a každý jeho prvek otestovat.

### Příklad

Na příkladě v Pascalu si ukážeme reprezentaci množiny celých čísel z rozmezí od −100 do 100. Nejprve definujeme konstanty vymezující povolený rozsah hodnot a deklarujeme pole pro uložení vlastní množiny:

```
const D = -100;     {dolní mez}
      H = 100;      {horní mez}
var   M: array [D..H] of boolean;
```

Následuje ukázka, jak lze realizovat jednotlivé základní operace s takto reprezentovanou množinou. Počáteční inicializaci množiny M zajistíme provedením příslušných dosazovacích příkazů. Nechť M obsahuje čísla −46, 3, 12 a 77:

```
for I := D to H do M[I]:=false;
                  {M zatím představuje prázdnou množinu}
M[-46] := true;
M[3]   := true;
M[12]  := true;
M[77]  := true;  {množina M obsahuje čtyři prvky}
```

Přidání čísla $X$ do množiny M provedeme rovněž přímým dosazením (přesněji: $X$ je proměnná typu integer, momentální hodnotu proměnné $X$ přidáme do množiny M):

```
M[X] := true;
```

Vypuštění čísla $X$ z množiny M je obdobné:

```
M[X] := false;
```

Test, zda je číslo $X$ obsaženo v množině M, se provádí jednoduše otestováním hodnoty $M[X]$:

```
if M[X] then ... {je obsaženo}
       else ... {není obsaženo}
```

Provedení *AKCE* s každým prvkem množiny M je trochu složitější, je nutné otestovat všechny složky pole M:

```
for I:=D to H do
  if M[I] then AKCE(I);
```



## 6.2 Seznam prvků

Reprezentace množiny výčtem jejích prvků ve tvaru seznamu je asi to první, co každého napadne. Na rozdíl od skutečných seznamů nám v tomto případě nezáleží na pořadí jednotlivých prvků. Navíc musíme dodržovat zásadu, že každý prvek je v seznamu obsažen nejvýše jednou (množina nemůže obsahovat dva stejné prvky). Seznam může být realizován v poli, pokud známe horní odhad maximálního možného počtu prvků v množině. Vedle vlastního pole budeme při psaní programu potřebovat ještě jednu celočíselnou proměnnou udávající aktuální počet prvků v množině (tj. velikost obsazené části pole). Nemáme-li žádný horní odhad počtu prvků množiny, musíme použít nějakou jinou datovou strukturu. Typicky se v této situaci používá lineární spojový seznam dynamicky alokovaných záznamů.

Nový prvek se do seznamu vkládá jednoduše na konec (v případě polí) nebo na začátek (u lineárních spojových seznamů). Vložení nového prvku do množiny má konstantní časovou složitost, pokud předem víme, že v množině tento prvek dosud není obsažen, a nemusíme tudíž procházet celým seznamem, abychom tuto skutečnost zkontrolovali. Jinak by mělo vkládání prvku lineární časovou složitost způsobenou náročností tohoto testu. Podobně vypuštění prvku ze seznamu bude mít konstantní časovou složitost tehdy, jestliže již máme k dispozici odkaz na vypouštěný prvek a nemusíme ho v seznamu vyhledávat podle hodnoty. Vypouštění se provádí tak, že se hodnota uložená v místě vypouštěného prvku přepíše hodnotou z posledního prvku seznamu (při reprezentaci pomocí pole) nebo z prvního prvku seznamu (u spojových seznamů) a ten se pak snadno zruší. Není tedy třeba pracně posouvat všechny prvky pole umístěné za rušeným prvkem nebo procházet spojovým seznamem od začátku a hledat předchůdce rušeného uzlu. Ostatní operace s množinami reprezentovanými seznamem prvků mají lineární časovou složitost vzhledem k aktuálnímu počtu prvků množiny (je nutné projít seznamem prvků).

### Příklad

Ukážeme si základní operace s množinou reprezentovanou seznamem prvků v poli. Budeme chtít reprezentovat množinu celých čísel, která obsahuje nejvýše 100 prvků:

```
const Max = 100; {maximální počet prvků v množině}
var   M: array [1..Max] of integer;
      Pocet: 0..Max; {počet prvků množiny M,
                      hodnota Pocet=0 představuje
                      prázdnou množinu}
```

Inicializace množiny M se opět provádí přímým dosazením — nechť M obsahuje čísla −46, 3, 12 a 77:

```
M[1] := -46;
M[2] := 3;
M[3] := 12;
M[4] := 77;
Pocet := 4;
```

Při přidávání čísla $X$ do množiny M (přesněji: $X$ je proměnná typu integer, momentální hodnotu proměnné $X$ přidáváme do množiny M) je dobré kontrolovat, zda se nové číslo do M ještě vejde:

```
Pocet := Pocet + 1;
if Pocet > Max then
                  begin
                  Pocet := Pocet - 1;
                  Error {pokus o překročení max.
                         povoleného počtu prvků
                         množiny M}
                  end
               else M[Pocet] := X;
```

Vypuštění čísla $M[J]$ z množiny M (již bylo zjištěno, že vypouštěné číslo má index $J$):

```
M[J] := M[Pocet];
Pocet := Pocet - 1;
```

Provedení *AKCE* s každým prvkem množiny M je jednodušší než v případě uvedeném v kap. 6.1, zde můžeme v cyklu procházet pouze prvky skutečně obsažené v M:



```
for I:=1 to Pocet do
  AKCE(M[I]);
```

Poslední zkoumanou akcí je test, zda je číslo $X$ obsaženo v množině M ($X$ je proměnná typu integer). Pokud je tam obsaženo, máme také určit jeho index. K tomu použijeme některý z postupů vyhledávání prvků v poli, které jsou uvedeny v kap. 5.2.

### 6.3 Uspořádaný seznam prvků

Seznam prvků, který reprezentuje množinu, můžeme udržovat uspořádaný podle hodnot klíče (například vzestupně). Tento uspořádaný seznam pak opět naprogramujeme pomocí pole nebo pomocí spojového seznamu. Oproti seznamu neuspořádanému se některé operace budou nyní provádět rychleji a jiné zase pomaleji. Volba vhodnější reprezentace množiny bude proto záviset na tom, jaké operace s množinami budou v našem konkrétním programu převažovat.

Přidání prvku do uspořádaného seznamu má lineární časovou složitost. Nejprve totiž musíme nalézt správné místo v seznamu, kam nový prvek patří, a tam ho pak zapojit. Při realizaci seznamu v poli je nutné odsunout všechny prvky ležící za nově vkládaným (což u seznamu délky $N$ představuje řádově $N$ elementárních operací), v případě spojového seznamu zase musíme postupně procházet seznamem a hledat správnou pozici nového prvku (rovněž řádově $N$ operací).

Vypuštění prvku si zachová konstantní časovou složitost pouze v případě, že je uspořádaný seznam uložen jako obousměrný lineární spojový seznam a známe přímo ukazatel na vypouštěný prvek (nemusíme ho v seznamu hledat podle hodnoty). Pokud pracujeme s polem, bude mít vypuštění prvku lineární složitost, neboť všechny větší prvky se musí v poli posunout. Test, zda je číslo $X$ obsaženo v množině M, a určení jeho polohy se naopak výrazně zrychlí, jestliže použijeme pole. K vyhledávání v uspořádaném poli použijeme metodu binárního vyhledávání, s kterou jsme se seznámili v kap. 5.2. Tato metoda má časovou složitost $O(\log N)$.

**Příklad**

Reprezentace množiny celých čísel vzestupně uspořádaným polem. Množina bude obsahovat nejvýše 100 prvků.

```
const Max = 100;
var   M: array [1..Max] of integer;
      Pocet: 0..Max;
```

Přidání čísla $X$ do množiny M. Předpokládejme, že již víme, že hodnota $X$ patří podle velikosti na pozici s indexem $j$ (pozice se zjistí nejlépe binárním vyhledáváním — viz dále):

```
if Pocet < Max then
  begin
  for i:=Pocet downto j do M[i+1] := M[i];
  M[j] := X;
  Pocet := Pocet+1;
  end
else
  Error;  {pokus o překročení max. povoleného
           počtu prvků množiny M}
```

Vynechání čísla s indexem $j$ z množiny M (vypouštěné číslo již bylo vyhledáno, známe jeho pozici):

```
if Pocet > 0 then
  Pocet := Pocet-1;
  for i:=j to Pocet do M[i] := M[i+1];
else
  Error;  {už není co vybírat}
```

Test, zda je číslo $X$ obsaženo v množině M. Pokud ano, určit jeho index (binární vyhledávání — viz také kap. 5.2):

```
i := 1;       {dolní mez}
j := Pocet;   {horní mez}
repeat
  k := (i+j) div 2;   {index prostředního prvku}
  if X > M[k] then i := k+1
              else j := k-1
until (M[k] = X) or (i > j);
if M[k] = X then ... {je obsaženo, X = M[k]}
            else ... {není obsaženo}
```



**Příklad**

Reprezentujte množinu celých čísel, neznáte-li předem horní odhad počtu jejích prvků. Lze očekávat, že počet prvků množiny bude velmi velký. Množina bude jednou vytvořena a pak už do ní nebudou další prvky přidávány ani z ní vypouštěny. Častokrát se ale bude testovat, zda nějaká hodnota je prvkem množiny.

Volbu vhodné reprezentace založíme na následujících úvahách. Neznáme omezení počtu prvků v množině, takže musíme použít spojový seznam. Prvky nebudou z množiny vypouštěny, proto stačí seznam jednosměrný (obousměrný seznam je výhodný právě pro operaci vypouštění). Do seznamu se bude málo vkládat, ale hodně se v něm bude vyhledávat. Bude tudíž asi výhodnější použít seznam uspořádaný. Oproti neuspořádanému seznamu bude vkládání prvku obtížnější (lineární místo konstantní časové složitosti). Vyhledávání zůstane sice v průměru lineární, ale díky uspořádání hodnot v seznamu se mnohdy výrazně zkrátí, což u častokrát opakovaného procházení dlouhého seznamu může znamenat významnou úsporu času.

Datová reprezentace:

```
type Uk = ^Uzel;
     Uzel = record
            Info: integer;
            Dalsi: Uk
            end;
```

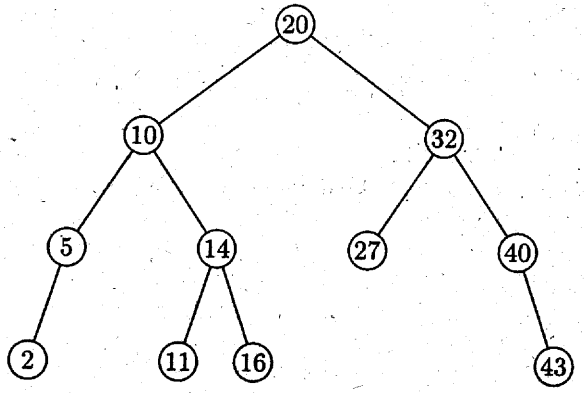
Test, zda je hodnota $X$ obsažena v množině M. Pokud ano, určit odkaz na příslušný prvek:

```
function Hledej (M: Uk; X: integer): Uk;
{ M - začátek spoj. seznamu reprezentujícího množinu
  X - hledaná hodnota
  funkční hodnota - ukazatel na prvek seznamu
                    s hodnotou X nebo nil, pokud X
                    v seznamu M není obsažen        }
```

```
var Nenalezen: boolean;
begin
  Nenalezen := true;
  while (M<>nil) and Nenalezen do
    if M^.Info = X then  {nalezen, konec procházení}
      begin Hledej:=M; Nenalezen:=false end
    else if M^.Info > X then   {není tam, konec}
      begin Hledej:=nil; Nenalezen:=false end
    else   {zkusit další prvek}
      M:=M^.Dalsi;
  if Nenalezen then Hledej:=nil
    {došli jsme až na konec seznamu a X tam není}
end;  {Hledej}
```

## 6.4 Binární vyhledávací strom

Binární vyhledávací strom je trochu složitější datová struktura používaná pro ukládání a vyhledávání údajů. Je to binární strom (vysvětlení pojmu binární strom naleznete v kap. 2.2), v jehož uzlech jsou umístěna data. Přitom pro uspořádání hodnot uložených ve stromu platí následující pravidlo: Je-li ve vrcholu $V$ uložena hodnota $X$, pak všechny uzly v levém podstromu vrcholu $V$ obsahují hodnoty menší než $X$ a všechny uzly v pravém podstromu vrcholu $V$ obsahují hodnoty větší než $X$.




*Obr. 11* Binární vyhledávací strom



Vyhledání vrcholu s hodnotou $X$ v daném binárním vyhledávacím stromě je snadné. Začneme v kořeni stromu a postupujeme směrem k listům na základě porovnání hodnoty uložené v uzlu stromu s hodnotou $X$. Je-li $X$ menší, postupujeme do levého následníka, je-li $X$ větší, postupujeme vpravo. Prohledávání končí buď nalezením uzlu s hodnotou $X$, nebo zjištěním, že v potřebném směru následník neexistuje, a že tedy hodnota $X$ není ve stromě uložena. Počet provedených porovnání je tedy menší nebo roven výšce stromu. Algoritmus si ukážeme ještě jednou v Pascalu:

```
type Uk = ^Uzel;
     Uzel = record
              Info: integer;
              L,P : Uk        {levý a pravý následník}
            end;

function Hledej (M: Uk; X: integer): Uk;
  { M - ukazatel na kořen binárního vyhledávacího stromu
    X - hledaná hodnota
    funkční hodnota - ukazatel na uzel stromu s hodnotou X
                      nebo nil, pokud X není ve stromě    }
  var Nenalezen: boolean;
  begin
    Nenalezen := true;
    while (M<>nil) and Nenalezen do
      if M^.Info = X then   {nalezen, konec procházení}
        Nenalezen:=false
      else if M^.Info > X then   {pokračování vlevo}
        M:=M^.L
      else
        M:=M^.P;                 {pokračování vpravo}
    Hledej:=M
  end;  {Hledej}
```

Při přidávání uzlu do binárního vyhledávacího stromu se postupuje stejně jako při vyhledávání. Je jen třeba zapamatovat si uzel, v němž procházení stromem skončilo a pod který máme zapojit nový uzel s přidávanou hodnotou. Maximální počet provedených elementárních operací je opět přímo úměrný výšce stromu:

```
procedure Pridej (var M: Uk; X: integer);
  { M - ukazatel na kořen binárního vyhledávacího stromu
    X - přidávaná hodnota
    Je-li ve stromu M hodnota X již obsažena, zůstane strom
    beze změn.                                              }
  var Nenalezen: boolean;
      U: Uk;       {zkoumaný uzel}
      Pred: Uk;    {předchůdce nového uzlu}
  begin
    Nenalezen := true;
    U:=M; Pred:=nil;
    while (U<>nil) and Nenalezen do
      if U^.Info = X then          {nalezen, konec procházení}
        Nenalezen:=false
      else if U^.Info > X then        {pokračovat vlevo}
        begin Pred:=U; U:=U^.L end
      else                            {pokračovat vpravo}
        begin Pred:=U; U:=U^.P end;
    if Nenalezen then              {připojit nový uzel za Pred}
      begin
        new(U);
        U^.Info:=X;
        U^.L:=nil; U^.P:=nil;
        if Pred=nil then                {nový kořen stromu}
          M:=U
        else if Pred^.Info>X then       {připojit vlevo}
          Pred^.L:=U
        else  {Pred^.Info<X}            {připojit vpravo}
          Pred^.P:=U
      end
  end;  {Pridej}
```

O něco komplikovanější je operace vypouštění uzlu z binárního vyhledávacího stromu. I ta má však časovou složitost úměrnou výšce stromu. Musíme opět vyhledat nejen uzel s rušenou hodnotou (označme si ho pro snazší vyjadřování se jako uzel $V$), ale i jeho předchůdce. Další postup závisí na tom, kolik má uzel $V$ následníků. Nemá-li žádného následníka, můžeme ho jednoduše zrušit a jeho předchůdce necháme ukazovat na `nil` místo na rušený uzel $V$ (na to nezapomínejte!). Pokud má uzel $V$



jednoho následníka, není podstatné zda levého nebo pravého, je postup také snadný. Uzel $V$ zrušíme a předchůdce uzlu $V$ bude nadále ukazovat místo na $V$ na následníka uzlu $V$.

Složitější situace nastává, pokud má uzel obsahující vypouštěnou hodnotu dva následníky. V takovém případě nemůžeme uzel přímo zrušit, neboť bychom neměli kam zapojit jeho následníky. Místo toho zrušíme pouze data uložená v tomto uzlu, nahradíme je daty z jiného uzlu stromu a ten pak vypustíme. Zbývá ukázat, který uzel stromu nám může takto dobře posloužit. Aby byly zachovány všechny nerovnosti, které musí v binárním vyhledávacím stromě platit, může to být buď uzel s největší hodnotou z levého podstromu uzlu $V$, nebo uzel s nejmenší hodnotou z pravého podstromu uzlu $V$. Zvolme si například první z uvedených možností. Uzel s největší hodnotou nalezneme v levém podstromu uzlu $V$ co nejvíce vpravo. Stačí tedy postoupit z uzlu $V$ do jeho levého následníka (tj. do kořene levého podstromu uzlu $V$) a z něj pak postupovat po pravých následnících tak dlouho, dokud je to možné. Hodnotu z takto získaného uzlu $S$ můžeme přemístit do uzlu $V$ a uzel $S$ ze stromu vypustit. Vypuštění bude snadné, neboť uzel $S$ nemá pravého následníka. Má tedy jen jednoho, nebo žádného následníka a takové uzly již umíme z binárního vyhledávacího stromu vypouštět.

Uvedený postup zapíšeme ještě jednou ve tvaru procedury v jazyce Pascal:

```
procedure Zrus (var M: Uk; X: integer);
  { M - ukazatel na kořen binárního vyhledávacího stromu
    X - rušená hodnota
    Není-li ve stromu M hodnota X obsažena, zůstane strom
    beze změn.                                              }
  var Nenalezen: boolean;
      U: Uk;          {zkoumaný uzel}
      Pred: Uk;       {předchůdce rušeného uzlu}
      S, PredS: Uk;   {skutečně rušený uzel v případě dvou
                       následníků a jeho předchůdce}
  begin
    Nenalezen := true;
    U:=M; Pred:=nil;
    while (U<>nil) and Nenalezen do
      if U^.Info = X then      {nalezen, konec procházení}
        Nenalezen:=false
```

```
    else if U^.Info > X then          {pokračovat vlevo}
      begin Pred:=U; U:=U^.L end
    else                              {pokračovat vpravo}
      begin Pred:=U; U:=U^.P end;
  if not Nenalezen then
    {X nalezeno, bude se rušit uzel U, jeho předchůdcem
     je uzel Pred}
    if U^.L=nil then          {přepojit U^.P, může to být nil}
      begin
      if Pred=nil then                {U je kořen}
        M:=U^.P
      else if Pred^.Info>X then       {U je levý následník}
        Pred^.L:=U^.P
      else                            {U je pravý následník}
        Pred^.P:=U^.P;
      dispose(U)
      end
    else if U^.P=nil then             {přepojit U^.L}
      begin
      if Pred=nil then                {U je kořen}
        M:=U^.L
      else if Pred^.Info>X then       {U je levý následník}
        Pred^.L:=U^.L
      else                            {U je pravý následník}
        Pred^.P:=U^.L;
      dispose(U)
      end
    else                              {U má dva následníky}
      begin
      S:=U^.L;
      PredS:=nil;
      while S^.P<>nil do
        begin PredS:=S; S:=S^.P end;
      U^.Info:=S^.Info;           {přenos hodnoty z S do U}
      if PredS=nil then           {S je přímý levý následník U}
        U^.L:=S^.L
      else                        {S je pravý následník PredS}
        PredS^.P:=S^.L;
      dispose(S)
      end
end;  {Zrus}
```



Jiný způsob naprogramování uvedených algoritmů s využitím rekurze můžete nalézt v knize [19]. Je však méně názorný a pro začátečníka hůře srozumitelný.

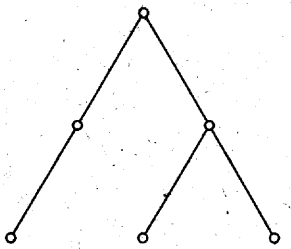
Časová složitost operací vkládání, vypouštění a testu náležení přímo závisí na výšce (tj. počtu hladin) binárního vyhledávacího stromu. Již z kap. 2.3 víme, že výška binárního stromu o $N$ vrcholech leží v rozmezí od $\log_2 N$ do $N$. Z hlediska časové efektivity je binární vyhledávací strom výhodnou reprezentací množiny tehdy, pokud je jeho výška úměrná $\log_2 N$ při uložení $N$ údajů. Každá z akcí vkládání, vypouštění nebo vyhledání vyžaduje totiž provést jeden průchod od kořene nejvýše k listu. V každém uzlu stromu se při tom vykoná konstantní počet elementárních operací, takže celá akce má složitost $O(\log N)$. To je v případě velkých $N$ výrazně lepší než lineární časová složitost uvedených operací u reprezentace množiny seznamem prvků.

Bude-li množina uložených údajů předem pevně dána a během výpočtu se nebude měnit přidáváním či rušením hodnot, není problém vybudovat tzv. **vyvážený vyhledávací strom** o $\log_2 N$ hladinách a výhodně v něm pak vyhledávat s logaritmickou časovou složitostí. Pokud se bude množina uložených dat během výpočtu měnit, hrozí nám, že se postupně výška stromu zvětší, a tím vzroste časová složitost všech operací (v nejhorším až k lineární složitosti). V takové situaci máme několik možností, co dělat. Nejjednodušší je nedělat nic a doufat, že všechno samo od sebe dobře dopadne a že k degeneraci stromu nedojde. S touto „strategií" máme kupodivu velkou šanci uspět. Nezaručuje nám sice stálé udržení logaritmické výšky binárního vyhledávacího stromu, ale zkušenosti z praxe ukazují, že při náhodné volbě vkládaných a rušených čísel se výška stromu většinou pohybuje kolem velmi příznivé hodnoty $1{,}4 \log_2 N$. Jinou možností je upravit algoritmy pro vkládání a vypouštění uzlů tak, aby strom zůstával stále dobře vyvážený a jeho výška byla stále přibližně logaritmická. Všechny akce pak budou mít zajištěnu časovou složitost $O(\log N)$. Podrobněji si o tomto řešení povíme v kap. 6.5. Konečně další možnou cestou je používat trochu jinou vyhledávací stromovou strukturu, která bude mít logaritmickou výšku stromu již jako nedílnou součást své definice. S jedním takovým typem stromů označovaným jako 2-3-stromy se seznámíme v kap. 6.6.

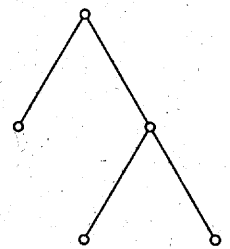
## 6.5 Vyvážené binární stromy

Pod pojmem vyvážený binární strom chápeme intuitivně takový binární strom, který má uzly rovnoměrně rozložené, nemá žádné „vyčnívající dlouhé větve", všechny listy jsou přibližně na stejné hladině (úplně to samozřejmě není možné zajistit). U stromu s $N$ uzly bude tímto vyvážením zajištěna výška $O(\log N)$. Díky tomu budou mít všechny operace založené na průchodu od kořene stromu k listu velmi příznivou logaritmickou časovou složitost.

Používá se více různých exaktních definic vyvážených stromů. Nejlepší možné vyvážení mají tzv. **dokonale vyvážené stromy**. Binární strom se nazývá dokonale vyvážený, jestliže pro každý uzel stromu platí, že počty uzlů v jeho levém a pravém podstromu se liší nejvýše o 1. Udržovat binární strom v dokonale vyváženém stavu při průběžném vkládání a vypouštění uzlů je však velmi obtížné. Proto se v praxi obvykle používá jiná definice vyváženosti a pracuje se s tzv. **AVL-stromy**. Binární strom nazýváme AVL-stromem, jestliže pro každý uzel stromu platí, že výšky jeho levého a pravého podstromu se liší nejvýše o 1. To je ve srovnání



*Obr. 12* Dokonale vyvážený strom



*Obr. 13* AVL-strom, který není dokonale vyvážený

s dokonale vyváženými stromy slabší požadavek. Každý dokonale vyvážený strom je již automaticky také AVL-stromem, ale opačné tvrzení neplatí. Výhodou AVL-stromů je, že se s nimi mnohem lépe pracuje a přitom postačují pro zajištění logaritmické výšky stromu. Lze totiž snadno dokázat, že každý AVL-strom o $N$ vrcholech má výšku nejvýše $2\log_2 N$ (důkaz tvrzení najdete např. v [13]).



Pro ukládání a opětovné vyhledávání údajů můžeme tedy použít binární vyhledávací strom, který splňuje podmínku AVL-vyvážení. Použijeme ho tehdy, pokud potřebujeme mít zaručenu za všech okolností nejvýše logaritmickou časovou složitost operací vyhledávání, vkládání i vypouštění prvku. V literatuře jsou popsány algoritmy pro vkládání uzlu do takového stromu a pro vypouštění uzlu z něj, které všechny potřebné vlastnosti binárního vyhledávacího AVL-stromu zachovávají. Přitom obě tyto operace mají časovou složitost $O(\log N)$. Pro jejich efektivní realizaci je jen zapotřebí mírně doplnit použitou datovou strukturu. Do každého uzlu stromu přidáme navíc jednu „technickou" položku, která bude nabývat jedné ze tří možných hodnot $-1, 0, 1$. Tato položka v každém uzlu udává, jak se liší výška jeho levého a pravého podstromu. Vlastní algoritmy zde nebudeme rozepisovat, neboť jsou dosti dlouhé a komplikované a jsou dobře popsány v dostupné literatuře. V knihách [13] a [16] najdete jejich popis a vysvětlení, v [19] jsou dokonce naprogramovány v Pascalu.

## 6.6 2-3-stromy

Další datová struktura, která nám zajistí logaritmickou časovou složitost všech základních operací s množinami (tj. testu náležení, vkládání a vypouštění prvku), se nazývá 2-3-strom. V odborné literatuře ji najdete také pod označením binární B-strom, neboť jde o speciální případ obecnější struktury, tzv. B-stromu. Představuje jinou cestu, jak lze zajistit vyváženost vyhledávacího stromu, zcela odlišně od myšlenky AVL-vyvážení. Definici obecných B-stromů i popis základních algoritmů prováděných nad nimi můžete nalézt například v [16].

Ve 2-3-stromech rozlišujeme uzly dvou typů. Zvláštní význam mají listy, tj. uzly bez následníků. Pouze v nich jsou umístěna uložená data. Všechny ostatní uzly (tzv. vnitřní uzly) jsou jiného typu a slouží pouze k vytvoření vyhledávací struktury 2-3-stromu. Každý vnitřní uzel obsahuje dva klíče pro vyhledávání a tři ukazatele na následníky:

```
type Uk = ^Uzel;
     Uzel = record
              K1, K2: integer;
              U1, U2, U3: Uk
            end
```

Struktura 2-3-stromu je definována následujícími dvěma pravidly:

1. Každý vnitřní uzel stromu má 2 nebo 3 následníky (odtud je také odvozen název stromu).

2. Všechny listy stromu jsou ve stejné výšce (na stejné hladině).

Potřebujeme-li uložit $N$ údajů, bude mít 2-3-strom $N$ listů. Nyní nás bude zajímat výška, tj. počet hladin, takového 2-3-stromu s $N$ listy. Označme si tuto výšku symbolem $H$. Pokud by měl každý uzel 2-3-stromu pouze dva následníky, obsahovaly by jeho jednotlivé hladiny postupně 1, 2, 4, 8, ..., $2^{H-1}$ uzly. Pro počet jeho listů by proto platila rovnost $2^{H-1} = N$. Jestliže bude mít naopak každý uzel 2-3-stromu tři následníky, v jednotlivých hladinách stromu bude ležet $1, 3, 9, 27, \ldots, 3^{H-1}$ uzlů. Maximální počet listů v 2-3-stromu výšky $H$ je tudíž $N = 3^{H-1}$. Pro počet listů $N$ obecného 2-3-stromu výšky $H$ jsme tedy odvodili nerovnost $2^{H-1} \leqq N \leqq 3^{H-1}$. Odtud jednoduchou úpravou dostáváme $\log_3 N + 1 \leqq \leqq H \leqq \log_2 N + 1$ neboli $H = O(\log N)$.

Ke každému celému číslu $N$, $N > 1$, existuje 2-3-strom s $N$ listy. V mnoha případech dokonce není jednoznačně určen, je možné sestrojit více různých 2-3-stromů s $N$ listy. Například již pro $N = 6$ může mít kořen 2-3-stromu dva následníky a každý z nich tři své následníky — listy, nebo naopak. Libovolně velkou množinu dat lze reprezentovat 2-3-stromem. Tato skutečnost nemusí být každému zřejmá na první pohled. Vždyť třeba úplný binární strom s $N$ listy ve stejné výšce existuje pouze pro $N$ tvaru $2^k$. Důkaz existence 2-3-stromu pro libovolné $N$ je však velmi snadný, přímo vyplývá ze skutečnosti, že do každého 2-3-stromu lze přidat nový uzel tak, aby výsledná struktura zůstala 2-3-stromem.

Význam klíčů a ukazatelů ve vnitřních uzlech 2-3-stromu je obdobný jako v případě binárních vyhledávacích stromů. Klíč $K1$ udává maximální hodnotu uloženou v levém podstromu, na nějž ukazuje $U1$. Má-li uzel jen dva následníky, jeho klíč $K2$ není definován a všechny hodnoty dat větší než $K1$ se nacházejí v podstromu, na který ukazuje $U2$. Ukazatel $U3$ má pak hodnotu `nil` jako příznak, že uzel má jen dva následníky. Pokud má uzel tři následníky, klíč $K2$ obsahuje maximální hodnotu uloženou v prostředním podstromu ($U2$). V prostředním podstromu tedy najdeme údaje větší než $K1$ a menší nebo rovné $K2$. Ukazatel $U3$ ukazuje na třetí podstrom, kde jsou umístěny hodnoty větší než $K2$.

Operace vyhledání hodnoty ve 2-3-stromě probíhá obdobně jako v binárním vyhledávacím stromě. Postupujeme od kořene k listu a v každém



uzlu se na základě porovnání jeho klíčů s hledanou hodnotou rozhodujeme, kam postoupit dál. Počet provedených elementárních operací je úměrný výšce stromu, a je tedy třídy $O(\log N)$. Trochu náročnější je vkládání nových hodnot do 2-3-stromu a jejich vypouštění. Příslušné algoritmy představují vždy jeden průchod od kořene k listu (tj. vyhledání místa, kam se má nový uzel připojit nebo kde se nachází rušený uzel) a poté jeden průchod od listu zpět směrem ke kořeni. Při tomto zpětném průchodu se modifikuje struktura stromu a hodnoty klíčů v uzlech tak, abychom získali opět 2-3-strom. V nejhorším případě vede zpětný průchod až ke kořeni stromu, mnohdy je ale kratší (skončí, když už další změny stromové struktury nejsou zapotřebí). Časová složitost obou algoritmů je opět úměrná výšce stromu, a je tedy $O(\log N)$.

Popíšeme alespoň ve stručnosti, jak probíhá vložení nového uzlu do 2-3-stromu. Vyhledávacím průchodem od kořene k listu zjistíme, kam nový uzel patří. Označme jako $U$ ten vnitřní uzel stromu, na nějž by měl být nový list připojen. Pokud má uzel $U$ zatím jen dva následníky, jednoduše mu přidáme náš nový uzel jako třetího následníka a podle potřeby změníme hodnoty klíčů ve stromě. Změna uložených klíčů se jistě týká uzlu $U$, může se však promítnout i do vyšších hladin stromu. Jestliže měl uzel $U$ již tři následníky, nemůžeme mu čtvrtého přidat. Rozdělíme proto $U$ na dva uzly, původní $U$ a nový $U'$. Každý z nich dostane dva ze čtyř listů (tj. ze tří původních následníků uzlu $U$ a toho nového). Uzel $U'$ potom zapojíme podobným způsobem vedle uzlu $U$ pod téhož jejich předchůdce, tj. s případným „zdvojením" předchůdce. Takto postupujeme v zapojování tak dlouho, dokud je třeba. Pokud při tomto postupu dojdeme až k rozdvojení kořene stromu, vytvoří se nový kořen a výška stromu se tím zvýší o 1. Vypouštění uzlů z 2-3-stromu probíhá velmi podobně.

Detailnějšímu rozboru operací s 2-3-stromy se zde již nebudeme věnovat a zájemce odkážeme na odbornou literaturu, v níž je tato problematika dostatečně zpracována. Stručné vysvětlení myšlenek 2-3-stromů najdete například v [9], podrobnější rozbor doplněný o programy v Pascalu je v knize [19].

## 6.7 Halda

Kromě základních operací s ukládanými daty, které jsme studovali až doposud, potřebujeme někdy v našich programech provádět i jiné činnosti. V tomto odstavci se zamyslíme nad výhodností různých datových reprezentací, pokud vedle vkládání nových prvků potřebujeme opakovaně vyhledávat nejmenší uloženou hodnotu a odebírat nejmenší prvek z množiny.

Z již probraných reprezentací množiny je možné použít uspořádaný seznam. V něm stojí nejmenší prvek na prvním místě, takže je snadno přístupný v konstantním čase. Snadné je nejenom jeho vyhledání, ale i jeho vypuštění. Pokud je seznam realizován dynamicky jako lineární spojový seznam, je to otázka pouhého posunutí ukazatele na začátek seznamu. Konstantní čas na vypuštění nejmenšího prvku nám ale postačí i v případě, že je uspořádaný seznam uložen v poli. Není totiž potřeba posouvat všechny prvky jako v případě obecného vypouštění ze seznamu v poli, stačí evidovat si v jedné pomocné proměnné pozici aktuálního začátku seznamu. Tento způsob manipulace je podrobněji popsán při výkladu fronty v kap. 7.2.

V uspořádaném seznamu je poněkud pracnější vkládání nových prvků, které vyžaduje čas $O(N)$. Pokud bychom chtěli zrychlit vkládání, můžeme použít binární vyhledávací strom. Nalezení i vypuštění minimální hodnoty v binárním vyhledávacím stromu nám nebude činit žádné problémy, minimální prvek se nachází ve stromu co nejvíce vlevo. Stačí tedy vyjít z kořene a sestupovat vždy do levého následníka uzlu tak dlouho, až levý následník nebude existovat. Tam je uložen nejmenší prvek. Počet provedených elementárních operací je přitom roven výšce stromu, stejně jako při vkládání nového prvku. Oproti uspořádanému seznamu se nám tedy mírně zrychlilo vkládání, ale zpomalilo vyhledávání a vypouštění nejmenšího prvku.

Pro naše účely je nejvýhodnější použít jinou datovou strukturu, která má také tvar binárního stromu a nazývá se **halda** (heap). Halda zajišťuje nalezení minimálního prvku v konstantním čase, odebrání minima a přidání nového prvku pak v čase $O(\log N)$. Její výhodou je algoritmická jednoduchost a možnost snadné realizace v poli, známe-li horní odhad maximálního počtu prvků v haldě.

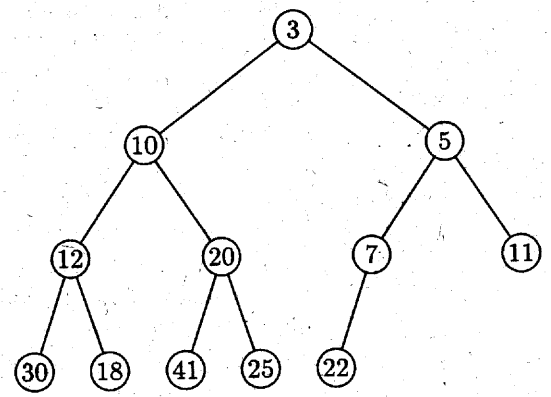
Halda je binární strom, ve kterém jsou splněny následující podmínky:



1. V každé hladině od první až do předposlední je maximální možný počet uzlů, tzn. v $k$-té hladině je $2^{k-1}$ uzlů.

2. V poslední hladině jsou všechny uzly umístěny co možná nejvíce „vlevo", tzn. procházíme-li uzly předposlední hladiny zleva doprava, nejprve má několik z nich (popř. žádný) dva následníky, pak může být (ale nemusí) jeden uzel s jedním následníkem a zbývající uzly předposlední hladiny následníky nemají.

3. Pro každý uzel platí, že hodnota v něm uložená je menší než hodnota uložená v jeho libovolném následníkovi.




*Obr. 14* Halda

Z třetí vlastnosti přímo vyplývá, že v kořeni haldy je umístěna nejmenší hodnota. Kořen stromu je přímo přístupný, takže operace nalezení minima má konstantní časovou složitost.

Nyní si ukážeme, jak se do haldy přidává nová hodnota. Nejprve musíme vytvořit jeden nový uzel a připojit ho do poslední hladiny co nejvíce vlevo, aby byl zachován správný tvar haldy (podle podmínek 1. a 2.). Do nového uzlu vložíme přidávanou hodnotu. Od této chvíle již nebudeme měnit tvar stromu, ale výměnami hodnot uložených v jednotlivých uzlech musíme zajistit jejich správné uspořádání (podle podmínky 3.). Začneme v nově připojeném uzlu $U$ a zkontrolujeme, zda je jeho hodnota větší než hodnota uložená v jeho předchůdci $P$. Pokud ano, je halda v pořádku a už s ní nic nemusíme dělat. V opačném případě je třeba vyměnit data uložená v těchto dvou uzlech a tím uspořádání

jejich hodnot napravit. Nově přidaná hodnota se tak přemístila do uzlu $P$ o hladinu výše a stejným způsobem budeme pokračovat v porovnávání od uzlu $P$ (tj. budeme porovnávat hodnoty nyní uložené v uzlu $P$ a v jeho předchůdci). Formování haldy končí v okamžiku, když se nově přidaná hodnota dostane postupnými výměnami buď do uzlu, jehož předchůdce již obsahuje hodnotu menší, nebo až do kořene. Počet provedených porovnání je tedy roven nejvýše celkové výšce haldy. Halda o $N$ vrcholech má výšku přibližně $\log_2 N$, takže přidání uzlu do haldy má časovou složitost $O(\log N)$.

Rozmyslete si dobře, proč je uvedený postup správný. Důležité je uvědomit si, že záměnou hodnot uložených v nějakém uzlu $U$ a jeho předchůdci $P$ „nic nepokazíme". Má-li totiž uzel $P$ ještě druhého následníka $R$, pak před výměnou obsahuje $R$ hodnotu větší než $P$. Výměnou hodnot v uzlech $U$ a $P$ se hodnota umístěná v $P$ nutně zmenší, takže správný vztah mezi uzly $P$ a $R$ zůstane zachován.

Dosti podobně probíhá vypouštění minimální hodnoty z haldy. Minimální hodnotu uloženou v kořeni smažeme. Haldu chceme zmenšit o jeden uzel. Musí to být uzel umístěný v poslední hladině co nejvíce vpravo, aby zůstal zachován správný tvar haldy. Jeho hodnotu $H$ však nesmíme ztratit, a proto ji v prvním okamžiku vložíme na uvolněné místo v kořeni haldy. Tím jsme získali strom o správném počtu uzlů se správnou množinou uložených údajů a se správným tvarem haldy. Zbývá pouze napravit vzájemnými výměnami uspořádání hodnot. Začneme od hodnoty $H$ v kořeni a postupujeme směrem k listům. Hodnotu nově umístěnou do kořene porovnáme s hodnotami obou jeho následníků. Je-li menší než oba následníci, halda je v pořádku a jsme hotovi. Jinak zaměníme hodnotu $H$ z kořene s menší z hodnot následníků. Tím jsou nerovnosti mezi uzly první a druhé hladiny napraveny a pokračujeme stejným způsobem o hladinu níže od uzlu, do kterého se právě přesunula hodnota $H$. Postupné výměny hodnot uzlů končí ve chvíli, kdy se hodnota $H$ dostane do místa, kde je menší než hodnoty následníků, nebo až do listu. Počet provedených porovnání určující časovou složitost popsaného algoritmu je tedy opět $O(\log N)$.

V programech bývá zvykem ukládat haldu do pole. Uzly haldy si očíslujeme postupně po hladinách vždy zleva doprava čísly od 1 do $N$. Kořen má tedy číslo 1, jeho následníci 2 a 3, na třetí hladině jsou uzly 4, 5, 6 a 7 atd. Tato čísla budou sloužit jako indexy pro uložení uzlů v poli.



Všimněte si, že zvolené očíslování má jednu velmi důležitou vlastnost: následníci uzlu s číslem $j$ mají čísla $2j$ a $2j + 1$. Po haldě se proto budeme velmi snadno pohybovat směrem nahoru i dolů jednoduchým přepočítáváním indexů.

Ukážeme si nyní algoritmy pro operace s haldou zapsané v programovacím jazyce Pascal. Předpokládáme, že halda nebude obsahovat více než 100 prvků, k její realizaci proto použijeme pole o 100 složkách. Prvky haldy budou pro jednoduchost celá čísla.

```
const Max = 100;              {max. počet prvků v haldě}

type  Halda = record
                Data: array[1..Max] of integer;
                                {vlastní data uložená v haldě}
                Pocet: 0..Max
                                {momentální počet prvků haldy}
              end;


function Min (var H: Halda): integer;
  {vrací hodnotu minimálního prvku v haldě H}
begin
  with H do
  if Pocet = 0 then
    Error
  else
    Min := Data[1]            {minimum je vždy v kořeni haldy}
end;  {Min}

procedure Pridej (var H: Halda; X: integer);
  {do haldy H přidává číslo X}
var j,p,d: integer;
    Pokracovat: boolean;
begin
  with H do
  if Pocet = Max then
    Error                     {překročen maximální povolený počet
                               prvků v haldě}
  else
    begin
```

```
    Pocet := Pocet+1;
    Data[Pocet] := X;
    j := Pocet;
    Pokracovat := j > 1;
    while Pokracovat do
      begin
      p := j div 2;
      if Data[j] < Data[p] then
        begin              {vyměnit hodnoty v uzlech j, p}
        d := Data[j];
        Data[j] := Data[p];
        Data[p] := d;
        j := p;            {posun v haldě na předchůdce}
        Pokracovat := j > 1
        end
      else               {halda je v pořádku, ukončit průchod}
        Pokracovat := false
      end
    end
end;  {Pridej}

procedure ZrusMin (var H: Halda);
  {vypouští z haldy H minimální prvek}
var j,n,d: integer;
    Pokracovat: boolean;
begin
  with H do
  if Pocet = 0 then
    Error                  {v haldě nic není, nelze vypouštět}
  else
    begin
    Data[1] := Data[Pocet];      {přesun dat do kořene haldy}
    Pocet := Pocet-1;
    j := 1;
    Pokracovat := 2 <= Pocet;
    while Pokracovat do
      begin
      n := 2*j;              {uzel n je levý následník uzlu j}
      if n < Pocet then      {existuje i druhý následník
                              uzlu j s indexem n+1}
        if Data[n+1] < Data[n] then n:=n+1;
```



```
      {nyní n je index menšího z následníků uzlu j}
      if Data[j] > Data[n] then
        begin              {vyměnit hodnoty v uzlech j, n}
        d := Data[j];
        Data[j] := Data[n];
        Data[n] := d;
        j := n;
        Pokracovat := 2*j <= Pocet
        end
      else               {halda je v pořádku, ukončit průchod}
        Pokracovat := false
      end
    end
end;  {ZrusMin}
```

Obě poslední procedury jsou zapsány tak, aby názorně ukazovaly, jak probíhají výměny prvků v haldě. Tentýž postup lze zapsat ještě o něco úsporněji. Ušetříme řadu přesunů v paměti, ale program bude pro někoho možná na první pohled méně srozumitelný:

```
procedure Pridej (var H: Halda; X: integer);
  {do haldy H přidává číslo X}
var j,p: integer;
    Pokracovat: boolean;
begin
  with H do
  if Pocet = Max then
    Error
  else
    begin
    Pocet := Pocet+1;
    j := Pocet;
    Pokracovat := j > 1;
    {X stále představuje hodnotu Data[j], která se však do
     pole Data průběžně neukládá a vloží se až na své
     definitivní místo po ukončení průchodu haldou}
    while Pokracovat do
      begin
      p := j div 2;
      if X < Data[p] then
        begin                  {hodnota X patří výše}
```

```
        Data[j] := Data[p];
        j := p;
        Pokracovat := j > 1
        end
      else
        Pokracovat := false    {halda je již v pořádku}
      end;
    Data[j] := X               {umístění hodnoty X do haldy}
    end
end;  {Pridej}

procedure ZrusMin (var H: Halda);
  {vypouští z haldy H minimální prvek}
var j,n,X: integer;
    Pokracovat: boolean;
begin
  with H do
  if Pocet = 0 then
    Error
  else
    begin
    X := Data[Pocet];
    Pocet := Pocet-1;
    j := 1;
    Pokracovat := 2 <= Pocet;
    {X stále představuje hodnotu přenesenou ze zrušeného
     listu do kořene haldy; tato hodnota se však do
     pole Data průběžně neukládá a vloží se až na své
     definitivní místo po ukončení sestupu haldou}
    while Pokracovat do
      begin
      n := 2*j;
      if n < Pocet then
        if Data[n+1] < Data[n] then n:=n+1;
      {nyní n je index menšího z následníků uzlu j}
      if X > Data[n] then
        begin                  {hodnota X patří níže}
        Data[j] := Data[n];
        j := n;
        Pokracovat := 2*j <= Pocet
        end
```



```
      else
        Pokracovat := false    {halda je již v pořádku}
      end;
      Data[j] := X             {umístění hodnoty X do haldy}
    end
end;  {ZrusMin}
```

Datová struktura halda se kromě jiného používá také v jednom ze známých algoritmů vnitřního třídění, v tzv. **třídění haldou** (heapsort). Úkolem je setřídit daných $N$ čísel podle velikosti od nejmenšího k největšímu. Jestliže již umíme pracovat s haldou, bude pro nás myšlenka algoritmu velmi jednoduchá. Z čísel, která máme setřídit, nejprve vybudujeme haldu. Začneme s primitivní haldou obsahující pouze jediné číslo (první ze tříděných čísel) a do této haldy postupně vložíme všechna zbývající tříděná čísla. Ve druhé etapě výpočtu celou haldu „rozebereme" tak, že z ní vždy odstraníme nejmenší prvek. Tím postupně dostáváme čísla uspořádaná podle velikosti od nejmenšího k největšímu. K setřídění $N$ čísel bude tedy nejprve $(N-1)$krát volána procedura *Pridej* a potom $N$krát funkce *Min* a procedura *ZrusMin*. Procedury *Pridej* a *ZrusMin* mají časovou složitost $O(\log N)$, jejich $N$ zavolání během třídění haldou proto představuje časovou složitost celého algoritmu $O(N \log N)$.

Haldu je možné vybudovat z daných $N$ čísel také jiným postupem, zdola od listů. Takový postup je na první pohled méně názorný než $N$-krát opakované volání procedury *Pridej*, jeho výhodou však je menší časová složitost. Celou haldu obsahující $N$ čísel je možné postavit v čase $O(N)$. Tento postup vytváření haldy můžete nalézt v knize [19].

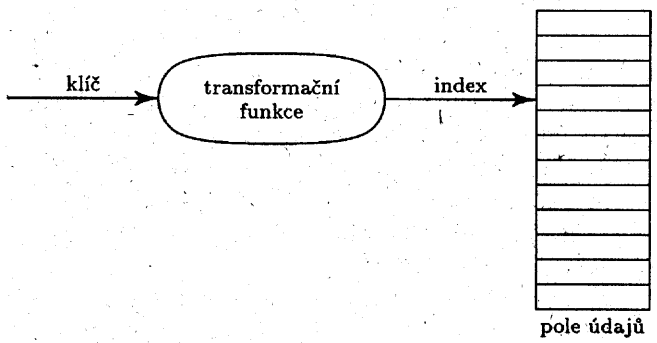
## 6.8 Rozptýlené tabulky

Reprezentace množiny hodnot pomocí tzv. rozptýlené tabulky představuje zcela odlišný přístup k řešení problému. S touto zajímavou metodou se v literatuře setkáte i pod různými jinými názvy. V knize [19] je použito označení „transformace klíče", ještě častěji se používají různé přepisy anglického termínu hashing nebo hash-tables (jako hašování, hešování, hešovací tabulky apod.).

Cílem metody je provádět operace testu náležení do množiny, vkládání i vypouštění prvku přímým indexováním pole, tedy v čase konstantním, nebo — jak uvidíme — spíše blízkém konstantnímu. Se stejnou

myšlenkou jsme se setkali již v kap. 6.1, kde jsme hovořili o reprezentaci množiny polem příznaků. Reprezentace polem příznaků sice poskytovala všechny požadované operace v konstantním čase, ale byla použitelná jen velmi omezeně: ukládané hodnoty musely být z předem známého ne příliš velkého rozmezí, aby bylo možné deklarovat patřičně velké pole přímo indexované ukládanými hodnotami. Nyní se pokusíme toto omezení odstranit a umožnit ukládání hodnot z rozsáhlejších rozmezí do polí menší velikosti. Navíc budeme požadovat, aby bylo možné metodu použít nejen pro ukládání samotných čísel, ale i pro ukládání celých záznamů a jejich vyhledávání na základě zvolené klíčové položky, jako tomu bylo u všech ostatních reprezentací.

Metoda je založena na použití tzv. **transformační funkce**, která každé ukládané hodnotě (resp. klíči v záznamu) přiřadí index, kam se má tato hodnota uložit do pole. Použité pole musí být samozřejmě dostatečně velké na to, aby se do něj vešly všechny ukládané prvky. Přípustné hodnoty klíčů však budou zpravidla z mnohem většího rozmezí, takže transformační funkce nebude prostá, více různým klíčům bude přiřazovat stejný index. Pokud bychom chtěli uložit zrovna takové dva prvky, kterým je přiřazen stejný index v poli, nastane tzv. **kolize**. Kolizím se obecně vyhnout nemůžeme a musíme mít nástroje k jejich řešení. Kolize snižují rychlost provádění operací s uloženými daty, proto se musíme snažit jejich počet minimalizovat. Budeme se tedy dále zabývat dvěma problémy: jednak volbou vhodné transformační funkce, aby ke kolizím nedocházelo příliš často, jednak postupem, jak vzniklé kolize řešit.




*Obr. 15* Transformační funkce



Základním požadavkem na transformační funkci je, aby rozptylovala co možná nejrovnoměrněji ukládané klíče v rozmezí indexů pole. V konkrétní úloze bude záležet na tom, co víme o ukládaných hodnotách. Pokud je například klíčem celé číslo a víme, že uložení jakéhokoli klíče je stejně pravděpodobné, pak nejjednodušší dobrou transformační funkcí je $f(K) = K \bmod N$, kde $K$ je ukládaný klíč a $N$ je počet prvků pole vyhrazeného pro ukládání hodnot (předpokládáme indexování pole od 0 do $N-1$). Jinak bychom ovšem volili transformační funkci třeba v situaci, že by klíčem bylo výrobní číslo nějakého výrobku a věděli bychom, že tři čtvrtiny používaných výrobních čísel končí dvěma nulami. Kdybychom použili pro ukládání informací o výrobcích pole o 100 prvcích a transformační funkci $\bmod 100$, většina pokusů o uložení dalšího výrobku by vedla ke kolizi.

Vzniklé kolize se řeší následovně. Chceme-li uložit novou hodnotu a transformací klíče dostaneme index takového místa v poli, které je již obsazeno, musíme pro nově vkládanou hodnotu nalézt náhradní umístění. Používané postupy lze rozdělit v zásadě do dvou tříd: buď se nové umístění hledá na jiné pozici v rámci téhož pole dat, nebo je pro kolidující záznamy vyhrazena zvláštní oblast v paměti. V prvním případě použijeme tzv. sekundární transformační funkci, která na základě indexu obsazeného místa v poli počítá index místa náhradního. Pokud by i to bylo již obsazeno, použije se sekundární transformační funkce opakovaně, až do nalezení volného místa. Konkrétní tvary používaných sekundárních transformačních funkcí lze nalézt v literatuře (např. [19]). Nejjednodušší je například posunutí o konstantu (modulo délka pole). V druhém případě se kolidující záznamy, jimž byl transformační funkcí přiřazen stejný index, řadí do pomocného seznamu. Na něj vede odkaz ze základního pole. Tento pomocný seznam bývá realizován dynamicky nebo v jiném poli a při vyhledávání je již procházen postupně prvek po prvku.

Analýza časové složitosti této metody je velmi komplikovaná a vychází z poznatků teorie pravděpodobnosti a matematické statistiky. V nejhorším případě může být metoda dosti špatná. Pokud bychom náhodou i přes dobrou volbu transformační funkce ukládali zrovna takové údaje, které by vždy vedly ke kolizi, bude výpočet velmi pomalý. V průměru je však metoda rozptýlených tabulek rychlá. Při dobré volbě rovnoměrně rozptylující transformační funkce závisí průměrný počet kolizí zejména na zaplněnosti tabulky. Tabulka musí být zvolena tak velká vzhledem

k počtu ukládaných dat, aby v ní stále zůstávala část volná. Ukazuje se, že až do zaplnění tabulky zhruba do poloviny ke kolizím téměř nedochází. Při zaplnění tabulky asi ze 75 % se průměrný počet kolizí, které musíme řešit při uložení jednoho nového záznamu, pohybuje kolem dvou a ještě asi do 90 % zaplnění tabulky je počet kolizí rozumně malý. Při větším zaplnění ovšem začne rychle narůstat.

## CVIČENÍ

**1.** V programu potřebujeme pracovat s množinou reálných čísel. Víme, že nikdy nebude obsahovat více než 50 prvků a že se bude často měnit, budou se do ní přidávat nová čísla a jiná se z ní budou vypouštět. Vyberte, které způsoby programové realizace takové množiny je možné použít, posuďte jejich výhodnost a některý vhodný způsob naprogramujte.

**2.** V programu potřebujeme pracovat s množinou reálných čísel. Víme, že nikdy nebude obsahovat více než 50 prvků, že se bude jen velmi málo měnit, ale zato se v ní bude často vyhledávat. Navrhněte a naprogramujte vhodnou datovou reprezentaci této množiny.

**3.** V programu potřebujeme pracovat s množinou reálných čísel. Nemáme k dispozici žádný horní odhad, kolik prvků bude obsahovat, její velikost bude záviset na konkrétních vstupních datech. Vyberte, které způsoby programové realizace takové množiny je možné použít, posuďte jejich výhodnost a některý vhodný způsob naprogramujte.



# 7 SEZNAMY PRVKŮ

V minulé kapitole jsme se zabývali otázkami vhodné reprezentace množin údajů a ukázali jsme si algoritmy pro ukládání, vypouštění a vyhledávání hodnot v množinách. Připomeňme si jednu důležitou vlastnost množin: vůbec nezáleží na pořadí, v jakém byly jednotlivé údaje do množiny vkládány. Toto pořadí se obvykle ani nezachovává v datové struktuře. Data se v množinách vyhledávají nebo vypouštějí podle zadané hodnoty klíče. Hodnota klíčové položky uložených dat jednoznačně určuje prvek množiny, v množině nemohou být uloženy zároveň dva stejné údaje (nebo údaje se stejnými klíči).

Druhou základní datovou strukturou pro ukládání údajů je tzv. **seznam**. Na rozdíl od množiny je seznam tvořen posloupností prvků s pevně stanoveným pořadím. Pořadí prvků je důležitým rysem seznamu a obvykle bývá neměnné. V seznamu se mohou nacházet opakovaně stejné údaje, což je jeho další důležitá odlišnost od množiny. Také do seznamu budeme chtít ukládat nové hodnoty, vypouštět z něj údaje, popř. i testovat, zda je daná hodnota v seznamu obsažena.

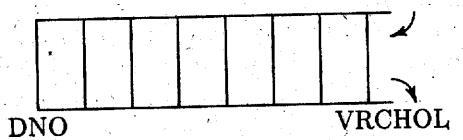
S pojmem seznam jsme se setkali již v kap. 6 jako s jednou z možných reprezentací množiny. Tam ale byly prvky množiny seřazeny do seznamu vlastně náhodně, na jejich pořadí vůbec nezáleželo a k ničemu se nevyužívalo. Nemuselo být také zachováváno beze změn. Díky tomu jsme mohli například velice rychle vypouštět prvky z množiny reprezentované seznamem v poli (viz příklad v kap. 6.2 — prvky pole se neposouvají, ale poslední hodnotu přemístíme na uvolněné místo po vypuštěném prvku). Pokud bychom však pracovali se skutečnými seznamy, v nichž je pořadí prvků důležité, musely by se po vypuštění prvku z prostředku seznamu všechny za ním umístěné prvky v poli posunout.

Pro programovou realizaci seznamu se nabízejí dvě základní možnosti: buď můžeme posloupnost prvků umístit do pole, nebo můžeme dynamicky generovat lineární spojový seznam. V případě pole musíme předem znát maximální možnou délku seznamu, abychom mohli příslušné pole vůbec deklarovat. U spojových seznamů takto omezeni nejsme, ale dynamická alokace má zase jiné nevýhody: je paměťově náročnější (do každého uzlu se s daty ukládá navíc ukazatel) a některé operace jsou i značně pomalejší (zejména vytvoření nového uzlu a jeho zrušení — v Pascalu volání standardních procedur `new` a `dispose`).

Se seznamy se pracuje obvykle tak, že se nové prvky přidávají pouze na některý z konců seznamu a odebírají se jen ze stejného nebo z opačného konce. Podle způsobu přidávání a odebírání dat se takové seznamy označují jako **zásobník** nebo **fronta**. U obou těchto speciálních druhů seznamů lze zvolit takovou programovou realizaci, aby přidání i vyjmutí prvku mělo konstantní časovou složitost. Zjištění, zda je jistá hodnota prvkem seznamu, má vždy lineární složitost — seznamem je nutné postupně projít a hodnotu vyhledat. Tato operace je však pro zásobník i pro frontu dosti netypická.

## 7.1 Zásobník

Zásobník je asi nejpoužívanějším typem seznamů. Je to takový seznam, u něhož jeden konec (tzv. vrchol zásobníku) slouží k přidávání i k odebírání prvků. Prvky se nemohou přidávat ani odebírat na druhém konci seznamu (označovaném jako dno zásobníku) ani nikde uprostřed. Kdykoli tedy chceme odebrat jeden prvek ze zásobníku, bude to ten, který byl do něj vložen jako poslední (je momentálně na vrcholu zásobníku). Zásobník proto bývá někdy označován jako struktura typu LIFO (z anglického last in — first out, tj. poslední dovnitř — první ven).




*Obr. 16* Zásobník

S praktickým použitím zásobníku se v naší knize setkáme ještě vícekrát. Používá se například při řešení úloh metodou „rozděl a panuj" k odložení informací o tom, které dílčí úkoly je ještě třeba vykonat (kap. 10), slouží jako základní datová struktura pro řízení průchodu do hloubky (kap. 8), používá se při zpracovávání aritmetických výrazů (kap. 12). Techniku zásobníku využívá i samotný mechanismus volání procedur a funkcí ve vyšším programovacím jazyce, kdy v paměti typu zásobník je přidělován prostor pro umístění lokálních proměnných volaných podprogramů. Zásobník s operacemi vkládání a vypouštění prvku se velice snadno programuje, z hlediska náročnosti obsluhy je to nejjednodušší typ



seznamu. Používá se proto i tehdy, potřebujeme-li dočasně uložit nějaké údaje a později je průběžně zpracovávat, aniž by nám záleželo na jejich pořadí (viz např. v kap. 9 algoritmus pro stanovení komponent souvislosti grafu nebo pro určení, zda je graf bipartitní).

Jestliže budeme zásobník programovat pomocí pole, bude první prvek pole vždy představovat dno zásobníku. Vedle vlastního pole dat použijeme ještě jednu proměnnou, která bude v každém okamžiku udávat index toho prvku pole, kde je momentálně vrchol zásobníku.

**Příklad**

Zásobník celých čísel s kapacitou pro uložení nejvýše 100 čísel:

```
const MaxZas = 100;
type  TypZas = record
        Zasobnik: array [1..MaxZas] of integer;
        Vrchol: 0..MaxZas;
      end;
```

Inicializace prázdného zásobníku:

```
procedure InicZasob(var Z: TypZas);
begin
  Z.Vrchol := 0
end;
```

Přidání jednoho čísla do zásobníku:

```
procedure VlozZasob(var Z: TypZas; X: integer);
begin
with Z do
  begin
  if Vrchol = MaxZas then
    Error   {zásobník zaplněn, X nelze vložit}
  else
    begin
    Vrchol := Vrchol + 1;
    Zasobnik[Vrchol] := X
    end
  end
end;
```

Vyjmutí jednoho čísla ze zásobníku:

```
function VezmiZasob(var Z: TypZas): integer;
begin
with Z do
  begin
  if Vrchol = 0 then
    Error   {zásobník prázdný, prvek nelze odebrat}
  else
    begin
    VezmiZasob := Zasobnik[Vrchol];
    Vrchol := Vrchol - 1
    end
  end
end;
```

Při realizaci zásobníku lineárním spojovým seznamem zcela vystačíme s obyčejným jednosměrným seznamem. Je jen třeba dobře si uvědomit, v jakém směru mají být jednotlivé prvky seznamu zřetězeny. Na začátek i na konec spojového seznamu se snadno vkládá nový uzel (pokud se nový prvek vkládá opakovaně na konec seznamu, je výhodné udržovat si stále pomocný ukazatel na dosud poslední prvek seznamu, za který se bude příště vkládat). Ze začátku spojového seznamu se také snadno vypouští první prvek. Vypuštění posledního prvku je ale o dost náročnější, je nutné projít celý seznam a vyhledat v něm předposlední prvek (předchůdce vypouštěného uzlu). Zásobník proto programujeme vždy tak, aby v pořadí první prvek spojového seznamu představoval vrchol zásobníku.

**Příklad**

Zásobník celých čísel v dynamické reprezentaci:

```
type Uk   = ^Uzel;
     Uzel = record
            Info: integer;
            Dalsi: Uk
            end;
var  Zasobnik: Uk;
```



Inicializace prázdného zásobníku:

```
procedure InicZasob(var Z: Uk);
begin
  Z := nil
end;
```

Přidání jednoho čísla do zásobníku:

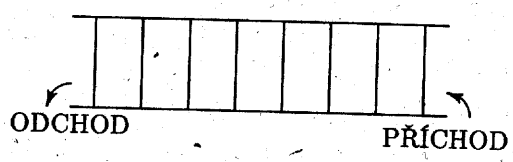
```
procedure VlozZasob(var Z: Uk; X: integer);
var P: Uk;        {pomocný ukazatel}
begin
  new(P);
  P^.Info := X;
  P^.Dalsi := Z;
  Z := P
end;
```

Vyjmutí jednoho čísla ze zásobníku:

```
function VezmiZasob(var Z: Uk): integer;
var P: Uk;        {pomocný ukazatel}
begin
if Z = nil then
  Error   {zásobník prázdný, prvek nelze odebrat}
else
  begin
  P := Z;
  Z := Z^.Dalsi;
  VezmiZasob := P^.Info;
  dispose(P)
  end
end;
```

## 7.2 Fronta

V seznamu označovaném jako fronta se pracuje s uloženými daty takovým způsobem, jaký odpovídá čekání lidí ve frontě v běžném životě. Přicházející se řadí za sebe, kdo dřív přišel, bude dříve obsloužen a odejde. Do seznamu dat typu fronta jsou proto na jednom konci („příchod") vkládány nově přidávané prvky, zatímco z druhého konce („odchod") jsou odebírány prvky vyřazované. V anglicky psané literatuře bývá fronta označována jako FIFO (first in — first out, tj. první dovnitř — první ven) nebo také FCFS (first come — first served, kdo první přijde, bude první obsloužen).




*Obr. 17* Fronta

Fronta se v programech používá tehdy, pokud potřebujeme dočasně odložit nějaká data a zachovat pro další zpracování jejich původní pořadí. Slouží jako základní datová struktura pro řízení průchodu do šířky (viz kap. 8). Její programová realizace v poli i dynamicky je o něco komplikovanější, než tomu bylo v případě zásobníku.

Frontu můžeme naprogramovat pomocí pole, pokud známe horní odhad její maximální možné délky (abychom mohli pole vůbec deklarovat a aby se do něj v každém okamžiku vešla všechna data uložená ve frontě). Budeme přitom potřebovat ještě dvě celočíselné proměnné, které budou udávat indexy těch prvků pole, kde je uložen první prvek fronty (tj. začátek fronty, odchod) a poslední prvek (konec fronty, příchod). Opakovanými operacemi vkládání a vypouštění prvků se budou hodnoty obou těchto proměnných stále zvyšovat, jak se bude směrem k vyšším indexům posouvat obsazená část pole. Tak by se mohlo po čase stát, že ačkoli aktuální délka fronty nikdy nepřekročí velikost pole, nebude možné vložit do fronty další údaj, neboť poslední prvek pole bude obsazen. Toto nebezpečí musíme nějak odstranit. Nabízejí se tři základní varianty řešení:

1. První, co by nás mohlo napadnout, je posouvat v poli všechny



prvky fronty o jedno místo směrem k nižším indexům po každém vypuštění prvku. Díky tomu bude začátek fronty neustále udržován v prvním prvku pole. Postup je to správný a na naprogramování jednoduchý, je však dost neefektivní.

2. Lepším řešením je posouvat v poli frontu pouze tehdy, když je to nutné, tzn. chceme-li do fronty vložit novou hodnotu a není kam (dosud poslední prvek fronty je uložen v posledním prvku pole). V takové situaci se tedy obsah pole přemístí najednou tak, aby fronta začínala v prvním prvku pole. Proti předchozímu řešení se přesuny hodnot v poli provádějí najednou o větší vzdálenost a méně často. Výpočet je proto celkově rychlejší.

3. Nejlepší samozřejmě bude data uložená v poli vůbec nepřesouvat. Představíme si, že pole je jakoby zatočené dokola a že po jeho posledním prvku následuje opět první prvek pole. Při vlastní programové realizaci pak budeme jenom patřičně přepočítávat indexy. Trochu složitějším se stane testování, zda fronta není prázdná (při odebírání prvku) nebo zda by její délka přidáním dalšího prvku nepřekročila kapacitu pole. Pro tento účel je vhodné zavést ještě jednu celočíselnou proměnnou, která bude udávat počet prvků ve frontě.

**Příklad**

Fronta celých čísel s kapacitou pro uložení nejvýše 100 čísel:

```
const MaxFr = 100;
type  TypFr = record
        Fronta: array [1..MaxFr] of integer;
        Zac,Kon: 1..MaxFr; {index prvního a posledního}
        Pocet:   0..MaxFr; {počet prvků ve frontě}
      end;
```

Inicializace prázdné fronty:

```
procedure InicFronta(var F: TypFr);
begin
with F do
  begin
  Pocet := 0;
  Zac := 1;
```

```
    Kon := MaxFr;
    end
  end;
```

Přidání jednoho čísla do fronty:

```
procedure VlozFronta(var F: TypFr; X: integer);
begin
with F do
  begin
  if Pocet = MaxFr then
    Error   {fronta zaplněna, X nelze vložit}
  else
    begin
    if Kon = MaxFr then Kon :=1
                   else Kon := Kon+1;
                                        {nový konec fronty}
    Fronta[Kon] := X;
    Pocet := Pocet+1
    end
  end
end;
```

Vyjmutí jednoho čísla z fronty:

```
function VezmiFronta(var F: TypFr): integer;
begin
with F do
  begin
  if Pocet = 0 then
    Error   {fronta prázdná, prvek nelze odebrat}
  else
    begin
    VezmiFronta := Fronta[Zac];
    if Zac = MaxFr then Zac := 1
                   else Zac := Zac+1; {nový začátek}
    Pocet := Pocet - 1
    end
  end
end;
```



K naprogramování fronty pomocí lineárního spojového seznamu nám opět postačí jednosměrný seznam. Podobně jako v případě zásobníku musíme jen dobře zvolit směr zřetězení prvků v seznamu. Jak jsme již uvedli v kap. 7.1, je dobré vyhnout se operaci vypuštění posledního prvku seznamu. Frontu je proto třeba reprezentovat tak, aby její začátek (odchod) byl na začátku seznamu a její konec (příchod) na konci seznamu. Každý uzel zařazený do fronty tedy obsahuje ukazatel na svého následníka. Je to obráceně, než bývá zvykem v běžném životě tam, kde čekající lidé nestojí přímo v řadě za sebou, ale udržují si pořadí podle doby příchodu (např. v čekárně u lékaře nebo u holiče). Tam si zpravidla každý pamatuje svého předchůdce, tj. toho, po kom přijde na řadu.

Z důvodu snadnějšího vkládání údajů do fronty je vhodné udržovat si kromě ukazatele na začátek fronty také odkaz na její poslední prvek. Není to sice nezbytné, ale pokud bychom přímý přístup k momentálně poslednímu prvku neměli, museli bychom při každém vkládání do fronty procházet celou frontu od začátku.

**Příklad**

Fronta celých čísel v dynamické reprezentaci:

```
type Uk    = ^Uzel;
     Uzel  = record
               Info: integer;
               Dalsi: Uk
             end;
     TypFr = record
               Zacatek: Uk;  {začátek fronty}
               Konec:   Uk;  {poslední prvek fronty}
             end;
var  Fr: TypFr;  {vlastní reprezentace fronty}
```

Inicializace prázdné fronty:

```
procedure InicFronta(var F: TypFr);
begin
  F.Zacatek := nil
end;
```

Přidání jednoho čísla do fronty:

```
procedure VlozFronta(var F: TypFr; X: integer);
var P: Uk;        {pomocný ukazatel}
begin
with F do
  begin
  new(P);
  P^.Info := X;
  P^.Dalsi := nil;
  if Zacatek = nil then
    Zacatek := P      {bude to jediný prvek ve frontě}
  else                {zařadit za dosud poslední prvek}
    Konec^.Dalsi := P;
  Konec := P          {nový konec fronty}
  end
end;
```

Vyjmutí jednoho čísla z fronty:

```
function VezmiFronta(var F: TypFr): integer;
var P: Uk;        {pomocný ukazatel}
begin
with F do
  begin
  if Zacatek = nil then
    Error   {fronta prázdná, prvek nelze odebrat}
  else
    begin
    P := Zacatek;
    Zacatek := Zacatek^.Dalsi;
    VezmiFronta := P^.Info;
    dispose(P)
    end
  end
end;
```

V některých programech se můžeme setkat i se zvláštními typy front, u nichž je způsob vkládání či vypouštění prvků proti klasické frontě nějak obohacen. První z těchto modifikací je tzv. **obousměrná fronta**. To je seznam, jehož konce jsou rozlišeny a prvky lze vkládat i vypouštět



na obou jeho koncích. Budeme-li chtít ukládat obousměrnou frontu do pole, budeme postupovat naprosto stejně jako v případě obyčejné fronty. Odliší se pouze akce „vlož na konec1“ a „vlož na konec2“, resp. „odeber z konce1“ a „odeber z konce2“. V případě efektivní realizace pomocí cyklického pole pak již jen stačí trochu doplnit způsob přepočítávání indexů. Pokud bychom reprezentovali obousměrnou frontu dynamicky pomocí lineárního spojového seznamu, bylo by vhodné použít obousměrný seznam, ve kterém lze na rozdíl od jednosměrného seznamu snadno přidávat i ubírat prvky na obou koncích.

Další variantou seznamu typu fronta je tzv. **prioritní fronta**. Prioritní frontou rozumíme seznam, který není uspořádán v pořadí příchodu prvků, ale podle jejich priorit. Každý prvek zařazovaný do seznamu má přidělenu zvláštní hodnotu, tzv. prioritu. Při vkládání do prioritní fronty není umístěn na konec fronty, ale „předběhne“ všechny prvky ve frontě, které mají prioritu nižší než on. Má-li více prvků stejnou prioritu, řadí se do fronty v pořadí jejich příchodu. Vyřazované prvky jsou odebírány jako v běžné frontě od začátku. Vždy je tedy odebrán prvek s nejvyšší prioritou (a z více prvků s maximální prioritou ten, který je ve frontě nejdéle).

Také prioritní frontu můžeme naprogramovat pomocí pole nebo lineárním spojovým seznamem. Pro umístění do pole platí všechno to, co bylo řečeno již o klasické frontě. Pouze operace vložení nového prvku bude komplikovanější, neboť je třeba vyhledat jeho správné umístění a všechny za ním stojící prvky pak v poli odsunout, aby se pro nový prvek uvolnilo místo. Nejlepší je postupovat od konce fronty a při hledání místa pro novou hodnotu hned prvky v poli posunovat. V dynamické reprezentaci prioritní fronty vystačíme s jednosměrným seznamem jako u fronty obyčejné. Vložení nové hodnoty do fronty vlastně představuje prosté zařazení prvku do uspořádaného lineárního spojového seznamu. Prioritní frontu můžeme výhodně realizovat také pomocí haldy, s níž jste se mohli seznámit v kap. 6.7. Operace přidání prvku do haldy a odebrání prvku z haldy musíme jen trochu upravit, aby bylo mezi prvky téže priority zachováno to pořadí, v němž byly do fronty zařazovány. Provedení této úpravy již ponecháme do cvičení.

## CVIČENÍ

**1.** Naprogramujte obousměrnou frontu celých čísel, vytvořte procedury pro přidání prvku na jeden a druhý konec fronty a pro odebrání prvku z jednoho a z druhého konce fronty.

**2.** Naprogramujte prioritní frontu celých čísel, v níž budou mít sudá čísla přednost před lichými. Vytvořte procedury pro přidání prvku do fronty a pro odebrání prvku z fronty.

**3.** Naprogramujte prioritní frontu celých čísel, v níž bude priorita prvků určena přímo samotnými uloženými čísly tak, že větší čísla mají přednost před menšími. Vytvořte procedury pro přidání prvku do fronty a pro odebrání prvku z fronty.

**4.** Naprogramujte prioritní frontu záznamů tvořených několika položkami. Jednou z položek každého záznamu bude celočíselný údaj o prioritě, ostatní položky záznamu si zvolte libovolně. Frontu realizujte pomocí haldy v poli.



# 8 PROHLEDÁVÁNÍ DO HLOUBKY A DO ŠÍŘKY

V této kapitole se budeme věnovat programovacím technikám umožňujícím systematicky procházet nějakým „vymezeným prostorem" a postupně navštívit všechna jeho místa. Jak uvidíme, tímto prostorem může být ledacos — třeba bludiště nebo hrací plán nějaké hry, graf představující silniční síť mezi městy, datová struktura tvaru stromu nebo obecného grafu nebo často jen množina možných stavů nějakého výpočetního procesu a přechodů mezi nimi. Techniky prohledávání prostoru do hloubky a do šířky se uplatňují ve všech těchto i v mnoha jiných případech, setkáte se s nimi v programech různého charakteru a z různých oblastí. Oba uvedené základní postupy mají stejný cíl, ale liší se celkovou organizací práce a pořadím, v jakém se jednotlivá místa zkoumaného prostoru procházejí. V důsledku toho se liší i jejich praktická použitelnost pro řešení konkrétních úloh a efektivita výsledných algoritmů a programů.

## 8.1 Procházení stromem a grafem

V kap. 2.2 jsme si připomněli pojmy strom a binární strom na teoretické úrovni, v kap. 5.4 jste se mohli seznámit se způsoby dynamické datové reprezentace stromů v programu. Každý vrchol stromu je představován jedním dynamicky generovaným exemplářem typu *Uzel*.

```
type T    = ... ;                {typ ukládaných dat}
     Uk   = ^Uzel;
     Uzel = record
              Info: T;           {uložená informace}
              L, P: Uk           {levý a pravý následník}
            end;
```

Binární strom je možné reprezentovat i statický v poli. Místo ukazatelů na následníky pak máme v každém uzlu uloženy dva indexy určující, kde jsou v poli umístěni následníci tohoto vrcholu. Nevýhodou statické reprezentace v některých programovacích jazycích (např. také v Pascalu) je nutnost deklarovat předem pole s pevně danou velikostí, tzn. nutnost znát předem dobrý horní odhad počtu vrcholů ve stromu.

```
const MaxVrch = 1000;                    {maximální počet vrcholů}
type  T = ... ;                          {typ ukládaných dat}
      Uzel = record
               Info: T;                  {uložená informace}
               L,P: 1..MaxVrch           {indexy následníků}
             end;
var   Strom: array[1..MaxVrch] of Uzel;  {uložení stromu}
      PocetVrch: 1..MaxVrch;             {počet vrcholů}
```

Předpokládejme nyní, že máme dán binární strom v dynamické reprezentaci, položka *Info* v jeho vrcholech je pro jednoduchost typu `integer` (jedno celé číslo). Naším úkolem bude vypsat v libovolném pořadí hodnoty *Info* ze všech uzlů stromu (nebo provést s každým uzlem nějakou jinou akci). Musíme tedy celý strom projít, postupně navštívit všechny jeho vrcholy a v každém z nich jednou vyvolat provedení jisté akce.

Nejjednodušší řešení využívá skutečnosti, že strom jako datová struktura má přirozeně rekurzivní charakter. Základem binárního stromu je jeho kořen, ten má nejvýše dva následníky a každý z nich je opět kořenem nějakého stromu. Průchod stromem s vypsáním hodnot všech vrcholů proto snadno popíšeme rekurzivním algoritmem: nejprve vypíšeme obsah kořene a poté necháme vypsat hodnoty všech vrcholů z jeho levého a pravého podstromu pomocí rekurzivního volání. V Pascalu může řešení úlohy vypadat třeba takto:

```
procedure Pruchod1(U: Uk);
  {průchod binárním stromem do hloubky pomocí rekurze.
   U je ukazatel na kořen zkoumaného stromu}
  begin
  if U <> nil then
    begin
    writeln(U^.Info);        {nebo jiná akce s uzlem U}
    Pruchod1(U^.L);
    Pruchod1(U^.P)
    end
  end; {procedure Pruchod1}
```



Právě popsaný algoritmus je nejjednodušším případem **průchodu do hloubky**. Všimněte si, jak procházení stromu vlastně probíhá, v jakém pořadí jsou jeho vrcholy navštěvovány a hodnoty uložené ve vrcholech vypisovány. Z každého vrcholu se jde po vypsání jeho hodnoty nejprve vlevo. Po průchodu celým jeho levým podstromem se do vrcholu vrátíme. Pak jdeme vpravo, projdeme celý jeho pravý podstrom a opět se do vrcholu vrátíme. Potom již tento vrchol definitivně opustíme a vracíme se z něj tam, odkud jsme do vrcholu přišli. To znamená, že strom je systematicky procházen zleva doprava, jedna jeho větev za druhou. Přitom se postupuje skutečně „do hloubky" stromu, vždy sestoupíme od kořene až do listu stromu, vrátíme se po stejné cestě jenom tolik, kolik je nezbytně třeba, abychom mohli zkoumat další cestu, a opět sestupujeme k dalšímu listu. Do každého vrcholu vstoupíme celkem až třikrát — nejdříve při sestupu od jeho předchůdce a pak při návratu z jeho levého a pravého podstromu (pokud je ovšem vrchol vůbec má). Požadovanou akci v uzlu však provádíme pouze jednou, vždy při prvním příchodu.

V principu je možné provádět akci v každém uzlu stromu také jindy — buď při druhém, nebo až při třetím příchodu do vrcholu. V programu takovou změnu zajistíme pouhou změnou pořadí tří řádků ve složeném příkazu procedury *Pruchod1*. Výše uvedený způsob procházení binárním stromem bývá označován jako **preorder**. Pokud akci v uzlu vykonáme mezi oběma rekurzivními voláními, tj. po druhém příchodu do vrcholu stromu, dostaneme postup zvaný **inorder**. Konečně poslední případ, kdy akci provádíme až těsně před definitivním opuštěním vrcholu, nazýváme průchod metodou **postorder**. Odlišnostem a významu těchto tří rozdílných způsobů procházení stromu do hloubky se budeme více věnovat v kap. 12, kde se binární stromy využijí k vnitřní reprezentaci aritmetického výrazu.

Algoritmus procházení binárním stromem do hloubky můžeme vyjádřit i bez použití rekurze. Mechanismus rekurzivního volání nahradíme vlastním zásobníkem (o zásobníku jsme psali více v kap. 7.1), do kterého si budeme odkládat informace o tom, které podstromy ještě musíme projít. Vyjdeme z kořene stromu, do zásobníku uložíme odkaz na jeho pravého následníka (pokud existuje) a přesuneme se do levého následníka. V něm zopakujeme totéž a podobně postupujeme stále dál tak dlouho, dokud to jde. Jestliže z nějakého právě navštíveného vrcholu nemůžeme pokračovat vlevo (nemá levého následníka), odebereme jeden vrchol ze

zásobníku a přesuneme se do něj. Stejný postup opakujeme tak dlouho, dokud se zásobník zcela nevyprázdní a nemáme již kam jít, tj. dokud neprojdeme celý strom.

Popsaný algoritmus se snáze vyjádří a naprogramuje, jestliže do něj proti výše uvedenému postupu přidáme navíc některé „zbytečné" přesuny dat do zásobníku a ze zásobníku. V každém kroku vyjmeme jeden vrchol ze zásobníku, provedeme s ním žádanou akci (např. vypíšeme jeho hodnotu) a do zásobníku vložíme jeho následníky, pokud je má. Má-li vrchol oba dva následníky, vložíme do zásobníku nejprve pravého a pak levého. Levý má být totiž navštíven dříve, a proto ho musíme umístit na vrchol zásobníku, aby byl hned v následujícím kroku ze zásobníku vyzvednut a zpracován. Dodržení tohoto pořadí ale není nutné, pokud nám nezáleží na tom, aby byl strom procházen postupně zleva doprava. Procházení stromem končí při vyprázdnění zásobníku. Celý postup můžeme zapsat schematicky, prozatím bez detailnějšího rozepsání, jak se pracuje se zásobníkem:

```
vlož do zásobníku kořen stromu
dokud zásobník není prázdný opakuj
    vlož do X údaj z vrcholu zásobníku a ze zásobníku
                                          ho odstraň
    proveď akci s vrcholem X
    má-li X pravého následníka, vlož ho do zásobníku
    má-li X levého následníka, vlož ho do zásobníku
konec cyklu
```

Při programování algoritmu musíme zvolit vhodnou reprezentaci zásobníku (viz kap. 7.1). Rozhodli jsme se pro dynamickou reprezentaci binárního stromu, abychom nebyli omezeni nutností znát předem horní odhad počtu jeho vrcholů. Z téhož důvodu budeme zde i zásobník reprezentovat dynamicky lineárním spojovým seznamem. Programové řešení v Pascalu bude nyní již snadné:

```
procedure Pruchod2(U: Uk);
  {průchod binárním stromem do hloubky pomocí zásobníku.
   U je ukazatel na kořen zkoumaného stromu}
  type UkZas = ^Zas;
       Zas   = record        {uzly tvořící zásobník}
                 Vrch: Uk;
                 Dalsi: UkZas
               end;
  var Zasob: UkZas;          {zásobník pro průchod do hloubky}

  procedure Vloz(U: Uk);     {vložení odkazu na vrchol U
                              do zásobníku Zasob}
    var Pom: UkZas;
    begin
      new(Pom);              {nový uzel}
      Pom^.Vrch := U;        {ukazuje na U}
      Pom^.Dalsi := Zasob;
      Zasob := Pom           {zapojení uzlu do Zasob}
    end;  {procedure Vloz}

  function Ven: Uk;          {vrací odkaz na vrchol vybíraný
                              ze zásobníku Zasob}
    var Pom: UkZas;
    begin
      Pom := Zasob;
      Zasob := Zasob^.Dalsi; {odstranění ze zásobníku}
      Ven := Pom^.Vrch;
      dispose(Pom)           {zrušit uzel}
    end;  {function Ven}

  begin  {procedure Pruchod2}
    Zasob := nil;            {zásobník je prázdný}
    if U <> nil then Vloz(U);{kořen stromu do zásobníku}
    while Zasob <> nil do    {dokud zásobník není prázdný}
      begin
      U := Ven;
      writeln(U^.Info);      {nebo jiná akce s uzlem U}
      if U^.P <> nil then Vloz(U^.P);
      if U^.L <> nil then Vloz(U^.L)
      end
  end;  {procedure Pruchod2}
```

Navštívit všechny uzly daného binárního stromu a provést v nich zvolenou akci (třeba vypsat položku *Info*) můžeme také při **průchodu stromem do šířky**. Procházením do šířky rozumíme procházení stromem po jeho vrstvách, po hladinách. Každá vrstva se prochází postupně zleva

doprava. Nejprve tedy navštívíme kořen, potom jeho levého a pravého následníka, pak jejich následníky v pořadí zleva doprava, dále všechny jejich následníky atd. Naposledy budou navštíveny listy ležící v největší vzdálenosti od kořene stromu.

Tak jako průchod do hloubky je řízen zásobníkem, k řízení průchodu do šířky používáme datovou strukturu typu fronta (viz kap. 7.2). Do ní si budeme opět ukládat informace, které podstromy ještě musíme projít a v jakém pořadí. Použití fronty místo zásobníku je to jediné, čím se průchod do šířky liší od průchodu do hloubky. Mechanismus fronty nám již sám zajistí správné pořadí návštěv jednotlivých vrcholů stromu po vrstvách. V každém kroku výpočtu vyjmeme jeden vrchol z fronty, provedeme s ním potřebnou akci (např. vypsání jeho hodnoty) a do fronty vložíme jeho následníky, pokud je má. Má-li vrchol oba dva následníky, vložíme do fronty nejprve levého a pak pravého následníka. Levý má být totiž navštíven dříve, a proto musí být dříve ve frontě. Procházení stromem končí v okamžiku vyprázdnění fronty.

Postup opět zapíšeme nejprve schematicky a pak ve tvaru procedury v Pascalu s frontou reprezentovanou lineárním spojovým seznamem.

```
vlož do fronty kořen stromu
dokud fronta není prázdná opakuj
    vlož do X údaj ze začátku fronty a z fronty ho odstraň
    proveď akci s vrcholem X
    má-li X levého následníka, vlož ho do fronty
    má-li X pravého následníka, vlož ho do fronty
konec cyklu
```

```
procedure Pruchod3(U: Uk);
  {průchod binárním stromem do šířky pomocí fronty.
   U je ukazatel na kořen zkoumaného stromu}
  type UkFr = ^Fr;
       Fr   = record      {uzly tvořící dynamicky frontu}
                Vrch: Uk;
                Dalsi: UkFr
              end;
  var ZacFr, KonFr: UkFr;
              {začátek a konec fronty pro průchod do šířky}

  procedure Vloz(U: Uk);   {vložení odkazu na vrchol U do fronty}
```



```
  var Pom: UkFr;
  begin
    new(Pom);                {nový uzel}
    Pom^.Vrch := U;          {ukazuje na U}
    Pom^.Dalsi := nil;
    if ZacFr = nil then      {fronta je prázdná}
      begin ZacFr:=Pom; KonFr:=Pom end
    else                     {zapojit na konec fronty}
      begin KonFr^.Dalsi:=Pom; KonFr:=Pom end
  end;  {procedure Vloz}

function Ven: Uk;            {vrací odkaz na vrchol vybíraný
                              z fronty}
  var Pom: UkFr;
  begin
    Pom := ZacFr;
    ZacFr := ZacFr^.Dalsi; {odstranění z fronty}
    Ven := Pom^.Vrch;
    dispose(Pom)             {zrušit uzel}
  end;  {function Ven}

begin  {procedure Pruchod3}
  ZacFr := nil;              {fronta je prázdná}
  if U <> nil then Vloz(U);{kořen stromu do fronty}
  while ZacFr <> nil do      {dokud fronta není prázdná}
    begin
    U := Ven;
    writeln(U^.Info);        {nebo jiná akce s uzlem U}
    if U^.L <> nil then Vloz(U^.L);
    if U^.P <> nil then Vloz(U^.P)
    end
end;  {procedure Pruchod3}
```

Stejné postupy průchodu do hloubky a do šířky, jaké jsme si ukázali pro binární stromy, můžeme uplatnit i při prohledávání obecných stromů nebo grafů. V případě grafů musíme jen dát pozor, abychom některý vrchol nenavštívili vícekrát. Je proto třeba použít vhodnou evidenci, který vrchol byl již navštíven (například pole logických hodnot přímo indexované čísly vrcholů). Grafům se budeme podrobněji věnovat v kap. 9, kde můžete nalézt příklady uplatnění průchodu grafem do hloubky nebo

do šířky v některých standardních grafových algoritmech, jako je určování komponent souvislosti, hledání nejkratší cesty v grafu, test bipartitnosti grafu apod. V ukázkových programech uvidíte, jak je možné průchod grafem vhodně naprogramovat.

## 8.2 Procházení stavovým prostorem

Řešení některých úloh spočívá ve zkoumání možných stavů, v nichž se může nacházet nějaký reálný systém, a ve sledování přechodů mezi nimi. Typické je to například při programování různých her a logických úloh, kdy stavy zkoumaného systému představují různé situace hry a přechody mezi nimi odpovídají jednotlivým tahům. Výpočet začíná v zadaném výchozím stavu (tj. v počáteční pozici hry). Úkolem je nalézt takovou posloupnost stavů a přechodů mezi nimi, která vede z výchozího stavu do některého koncového stavu. Koncových stavů může být i více, zpravidla nejsou zadány výčtem, ale je známa nějaká jejich charakteristická vlastnost. Při řešení úlohy hledáme obvykle jeden libovolný koncový stav (jedno kterékoli řešení úlohy). Někdy se ale požaduje zjistit všechny koncové stavy (nalézt všechna možná řešení).

Jako jednoduché příklady úloh tohoto typu nám dobře poslouží některé logické úlohy na šachovnici. Později si ukážeme i dosti odlišné úlohy, které se řeší stejnými metodami a které jsou mnohem bližší praxi. Půjde vlastně o všechny úlohy, v nichž musíme během cesty od počáteční situace k nalezení řešení projít přes několik míst, ve kterých je třeba zvolit jednu z více variant pokračování výpočtu. Přitom předem nevíme a nedokážeme si spočítat, která volba povede ke správnému řešení.

Naší první ukázkovou úlohou je úkol proskákat celou šachovnici šachovým koněm tak, aby kůň navštívil každé pole šachovnice právě jednou. Úlohu budeme řešit na čtvercové šachovnici velikosti $N \times N$ polí, počáteční pozice koně bude zadána na vstupu. Ve druhé úloze máme dáno výchozí a cílové pole na šachovnici a máme nalézt nejkratší cestu šachovým koněm mezi nimi. Než začneme úlohy řešit, připomeneme si ještě, jak se šachový kůň pohybuje. Jedním tahem se kůň posune o dvě pole ve směru svislém nebo vodorovném a zároveň o jedno pole kolmo na předchozí směr. Z pozice $(i, j)$ tedy můžeme provést některý z osmi možných tahů, a to na pozice $(i+1, j+2)$, $(i+2, j+1)$, $(i+2, j-1)$, $(i+1, j-2)$, $(i-1, j-2)$, $(i-2, j-1)$, $(i-2, j+1)$, $(i-1, j+2)$. Stojí-li kůň



u okraje šachovnice, nebo dokonce v jejím rohu, má samozřejmě možností méně.

V obou úlohách hledáme nějakou cestu koněm po šachovnici. Cesta se skládá z jednotlivých tahů. Po každém tahu se musíme rozhodnout z více možností, kam jít dál. Počet možných pokračování cesty je dán jednak momentální polohou koně na šachovnici, jednak požadavkem, že se na cestě nesmí žádné navštívené pole opakovat. V první úloze tento požadavek vyplývá přímo ze zadání, ve druhé úloze je zase zřejmé, že cesta s opakujícími se poli nebude nejkratší, a že ji proto můžeme předem vyloučit. Všechny možné cesty koně z daného výchozího pole si můžeme znázornit stromem, ve kterém vrcholy odpovídají jednotlivým stavům na šachovnici a hrany tahům koně. Stav je představován nejen momentální pozicí koně, ale i jeho dosavadní cestou. Kořen stromu představuje počáteční stav, kdy kůň stojí na svém výchozím poli a nemá ještě prošlou žádnou cestu. Kořen má až osm následníků (podle jednotlivých možných tahů koně z výchozí pozice), každý z nich má zase své následníky (kam lze dojít dvěma tahy) atd. Listy stromu odpovídají stavům, kdy kůň nemůže pokračovat nikam dál ve své cestě buď proto, že dosáhl požadovaného cíle, nebo proto, že ačkoli cíle nedosáhl, pravidla konkrétní řešené úlohy mu žádný další tah nepovolují. Tento strom všech možných cest je samozřejmě jen pomyslný, v uvedené podobě není uložen nikde v datových strukturách. Obvykle bývá velmi rozsáhlý, takže by to ani nebylo možné. Je vytvářen pouze po částech až během vlastního procházení.

Při řešení úlohy máme vlastně vyhledat ve stromě takový list, který odpovídá koncovému stavu. Zároveň máme určit, kudy do něj vede cesta z kořene. Charakterizace koncových stavů je u každé úlohy jiná. V případě naší první úlohy je koncovým stavem každý list v hloubce $N^2$, tzn. stav, do něhož vede cesta z kořene délky $N^2 - 1$ tahů. Na cestě se žádná poloha koně neopakuje, takže takovou cestou projde kůň celou šachovnici. U druhé úlohy je koncovým stavem každý list, který odpovídá umístění koně na zadaném cílovém poli. V tomto případě nás však nezajímá jeden libovolný koncový stav, nýbrž pouze ten s nejmenší hloubkou (hledáme nejkratší cestu).

K vyhledání jednoho nebo všech koncových stavů musíme provést průchod pomyslným stromem reprezentujícím stavový prostor úlohy. Strom můžeme prohledávat do hloubky nebo do šířky naprosto stejnými postupy, jaké jsme si ukázali v kap. 8.1 pro případ skutečných stromů.

uložených v datových strukturách. Jedinou odlišností bude to, že místo průchodu daným hotovým stromem si nyní budeme stavový strom po částech generovat souběžně s jeho prohledáváním. Před vyřešením našich dvou ukázkových úloh si ještě jednou popíšeme obě metody prohledávání stavového prostoru a porovnáme je z hlediska použitelnosti a efektivity.

Prohledávání do hloubky, nazývané také prohledávání s návratem nebo ještě častěji anglickým termínem **backtracking**, je metoda použitelná teoreticky zcela univerzálně. Jestliže postupně projdeme celý stavový prostor, rozhodně nemůžeme řešení úlohy minout (pokud ovšem vůbec existuje). Také obě naše úlohy lze řešit průchodem do hloubky. Z hlediska praktické použitelnosti programů založených na backtrackingu ale musíme být velmi opatrní. Metoda vede k programům s exponenciální časovou složitostí, a tedy použitelným pouze pro řešení malých úloh. Používáme ji proto spíše jen jako krajní řešení tam, kde nám nic jiného nezbývá, a ještě se snažíme různými způsoby výpočet urychlit (o tom více v kap. 8.3). Pokud bychom totiž v každém tahu volili jenom ze dvou možných pokračování a pokud by se listy stromu nacházely v hloubce přibližně $N$, bude mít stavový prostor úlohy velikost řádově $2^N$. V našich dvou úlohách na šachovnici bude v některých stavech více variant volby (bývá více možných tahů, v kořeni až 8, jinde až 7) a některé listy stromu leží v hloubce 64, což představuje hrozivý hrubý horní odhad časové složitosti algoritmu $7^{63}$ (tj. přibližně $10^{54}$).

Prohledávání do hloubky vlastně představuje zkoušení všech možností. Začneme ve výchozím stavu, určíme všechna možná pokračování výpočtu (přípustné tahy), nějak si je uspořádáme (můžeme libovolným způsobem) a postupně je budeme zkoušet provést. Provedeme první možný tah a ostatní si zapamatujeme. Zvoleným tahem se dostaneme do nějakého stavu $S$. Nejprve otestujeme, zda stav $S$ není koncový. Pokud je koncový, vykonáme potřebné akce: nalezené řešení si zapamatujeme nebo vytiskneme a pak celý výpočet buď ihned ukončíme (jestliže jsme hledali pouze jedno libovolné řešení), nebo v něm budeme normálně pokračovat (pokud hledáme všechna přípustná řešení úlohy). Určíme všechny následníky stavu $S$ a dále budeme postupovat stejně jako ve výchozím stavu. Jestliže ze stavu $S$ není možné nijak pokračovat v cestě (je to list), vrátíme se do jeho předchůdce na zkoumané cestě (odkud jsme do $S$ přišli) a tam budeme zkoušet provést další možné pokračování. Pokud jsme v některém stavu již všechny možnosti tahu vyzkoušeli a řešení jsme



dosud nenašli, opět se vrátíme o tah zpět. Výpočet končí buď přerušením prohledávání při získání koncového stavu (tj. při nalezení řešení úlohy), nebo ukončením průchodu celým stromem při návratu do kořene z jeho posledního následníka.

Pro programovou realizaci backtrackingu je velmi typické použití rekurze. To jsme ostatně viděli již v kap. 8.1 při průchodu binárního stromu do hloubky. Rekurzivní procedura dostane informaci o momentální situaci a dokončí řešení celé úlohy tak, že postupně vyzkouší v této situaci všechny možné tahy. Situace vzniklé těmito tahy zpracuje pomocí rekurzivního volání sebe sama. Nezapomeneme samozřejmě pokaždé nejprve otestovat, zda momentálně zpracovávaný stav není koncový. Naprogramování průchodu do hloubky si za malou chvíli ukážeme na naší úloze o cestě koně po šachovnici.

Prohledávání stavového prostoru do šířky spočívá v procházení stromu všech možných cest výpočtu po vrstvách. Bývá označováno také jako **algoritmus „vlny"** pódle názorné představy, že se z výchozího bodu rovnoměrně všemi směry šíří jakási vlna (jako třeba kruh na vodní hladině po vhození kamene do vody) a vrcholy jsou prohledávány v tom pořadí, jak jsou touto vlnou zasaženy. Procházení do šířky je určeno pouze pro takové úlohy, u nichž máme zajištěno, že některý koncový stav se ve stromu nachází v poměrně malé hloubce. V takovém případě vede k výrazně rychlejšímu řešení úlohy než prohledávání do hloubky, neboť z celého třeba i velmi rozsáhlého stavového prostoru se projde jen malý počet uzlů v horních, rozsahem menších hladinách stromu do nalezení prvního koncového stavu. Vhodná organizace výpočtu zpravidla umožňuje sestavit program s polynomiální časovou složitostí. Další výhodou průchodu do šířky je skutečnost, že má-li úloha více různých řešení, to nalezené je minimální z hlediska počtu provedených tahů. Tím je rozhodně výhodné, ať už je požadavek minimálního počtu tahů přímo součástí zadání úlohy, nebo ne. Průchod do šířky ovšem nemůžeme použít v úlohách, ve kterých může mít i nejkratší řešení příliš mnoho tahů, neboť by vedl k nereálným paměťovým nárokům programu. K jeho realizaci je totiž zapotřebí uchovávat najednou informaci o všech stavech právě procházené vrstvy stromu a počet uzlů na jedné hladině stromu může růst exponenciálně v závislosti na hloubce hladiny.

Prohledávání do šířky představuje souběžné zkoumání všech možných cest vedoucích z výchozího stavu směrem k cílovému tak dlouho,

dokud na některé z nich nenajdeme řešení. Výpočet začneme provádět ve výchozím stavu a určíme všechna možná jeho pokračování, tj. přípustné tahy. Postupně je všechny provedeme, vzniklé stavy z druhé hladiny stromu si zapamatujeme a otestujeme je, zda některý z nich není koncový. Pokud ano, máme řešení úlohy a výpočet ihned ukončíme. Pokud ne, vygenerujeme a zaznamenáme všechny stavy třetí hladiny jako všechna možná bezprostřední pokračování všech uložených stavů z druhé hladiny. Stejným způsobem postupujeme stále dál až do nalezení řešení.

Při řešení naší první ukázkové úlohy na šachovnici (proskákání celé šachovnice koněm) nelze průchod do šířky použít. Souběžné zkoumání všech možných cest koně z výchozího pole je paměťově naprosto nereálné. Zato naše druhá úloha (nejkratší cesta koněm mezi dvěma poli) se bude s výhodou řešit právě procházením do šířky. Mezi libovolnou dvojicí polí na šachovnici totiž existuje poměrně krátká cesta koněm. Například na klasické šachovnici $8 \times 8$ polí má tato cesta délku nejvýše 6 tahů. Navíc ani nemusíme zkoumat nezávisle na sobě všechny uzly prohledávacího stromu. Mnohé z nich můžeme ztotožnit, a tím výpočet ještě výrazně zrychlit, neboť nás zajímá pouze jedna cesta minimální délky a pro každé pole šachovnice je podstatné jenom to, kolik tahů koněm je vzdáleno od výchozího pole. Podrobněji se k tomuto řešení úlohy vrátíme za chvíli a ukážeme si i různé možnosti, jak ho lze naprogramovat. Teď ještě poznamenejme, že i tuto druhou úlohu by bylo teoreticky možné řešit průchodem do hloubky, avšak takové řešení by bylo velice pomalé a nešikovné. Postupně by se musely zkoumat všechny možné i velmi dlouhé cesty a z nich by se vybírala cesta s minimální délkou.

Na našich dvou příkladech si nyní ukážeme, jak se algoritmy průchodu do hloubky a do šířky mohou naprogramovat. Některé datové struktury budou mít oba programy společné. Šachovnici budeme reprezentovat maticí celých čísel $S[1..N, 1..N]$, do které budeme zaznamenávat průběh cesty koněm. Všechny prvky matice $S$ budeme inicializovat hodnotou $-1$, která bude značit dosud nenavštívené pole. Výchozí pole koně označíme hodnotou 0. Během procházení pak budeme do prvků pole $S$ umisťovat kladná čísla, která budou určovat, kolik tahů kůň vykonal k dosažení příslušného pole. Pro snadnou realizaci pohybu koně po šachovnici použijeme konstantní pomocné pole $Tah[1..8]$, jeho složkami budou záznamy tvořené dvěma celými čísly $d1$, $d2$. V poli $Tah$ budou uloženy přírůstky souřadnic na šachovnici pro osm možných směrů, jak může kůň táhnout.



Složka $d1$ udává vždy přírůstek první souřadnice a složka $d2$ přírůstek druhé souřadnice. Hodnoty pole $Tah$ tedy budou vypadat třeba takto (na pořadí nezáleží):

$$\begin{array}{ll} Tah[1].d1 = 1 & Tah[1].d2 = 2 \\ Tah[2].d1 = 2 & Tah[2].d2 = 1 \\ Tah[3].d1 = 2 & Tah[3].d2 = -1 \\ Tah[4].d1 = 1 & Tah[4].d2 = -2 \\ Tah[5].d1 = -1 & Tah[5].d2 = -2 \\ Tah[6].d1 = -2 & Tah[6].d2 = -1 \\ Tah[7].d1 = -2 & Tah[7].d2 = 1 \\ Tah[8].d1 = -1 & Tah[8].d2 = 2 \end{array}$$

První úlohu s nalezením cesty koně po celé šachovnici vyřešíme průchodem do hloubky. Šachovnice představovaná polem $S$ nám přitom bude sloužit k řízení průchodu a zároveň k ukládání nalezené cesty. Pole navštívená při právě zkoumané cestě budou mít kladnou hodnotu. Podle toho při prodlužování cesty poznáme, které pole bylo již navštíveno, a kam tedy nesmíme vstoupit podruhé. Při návratu zpět z neúspěšného pokusu o nalezení výsledné cesty ale nesmíme zapomenout ukládat do pole $S$ na opouštěné pozice opět hodnotu $-1$. Jestliže najdeme cestu délky $N^2 - 1$ tahů, výpočet ihned ukončíme. V poli $S$ pak bude uloženo nalezené řešení — výsledná cesta koněm. K naprogramování uvedeného postupu použijeme rekurzivní proceduru *Cesta*. Vstupními parametry této procedury budou souřadnice pole na šachovnici, kde momentálně stojí kůň. Všechny ostatní potřebné údaje, tzn. informace o dosavadním průběhu jeho cesty, budou uloženy v globálním poli $S$. Procedura tedy „převezme" cestu koně v jistém stavu a pokusí se ji všemi způsoby prodloužit a nalézt řešení. Přitom postupuje tak, že postupně vyzkouší všechny možné tahy koně z jeho momentální pozice na dosud nenavštívená pole na šachovnici a stavy vzniklé těmito tahy zpracuje pomocí rekurzivního volání. Výstupním parametrem procedury *Cesta* bude logická hodnota signalizující, zda se podařilo nalézt úspěšnou cestu délky $N^2 - 1$ tahů. Tento parametr slouží k okamžitému ukončení průchodu (zabrání zkoušení dalších variant cesty). Je také informací poskytovanou hlavnímu programu, jak řešení úlohy dopadlo. Dosti podobné řešení této úlohy si můžete přečíst také v knize [19].

Úlohu by bylo možné řešit i bez použití rekurze. Mechanismus rekurzivního volání bychom nahradili vlastním zásobníkem v programu stejným způsobem, jaký jsme si ukázali již v kap. 8.1 při průchodu binárním stromem do hloubky. Toto řešení si ponecháme do cvičení.

K řešení úlohy se vrátíme ještě jednou v kap. 8.3, kde se více zamyslíme nad jeho časovými nároky a nad možnostmi výpočet zrychlit.

---

```
program PruchodSachovnice;
{Nalezení takové cesty koněm po šachovnici, aby bylo každé
pole šachovnice navštíveno právě jednou.
Použitá metoda řešení: prohledávání s návratem, realizované
pomocí rekurzivní procedury.}

const N = 8;              {velikost strany čtvercové šachovnice}
      PocetKroku = 63;  { = N*N-1}
                {počet tahů koně tvořících výslednou cestu}
      PocetTahu = 8;  {počet různých tahů z jednoho pole}
type  Index = 1..N;   {typ indexu na šachovnici}

var S: array[Index,Index] of integer;          {šachovnice}
    Tah: array[1..PocetTahu] of
           record d1,d2:integer end;       {možné tahy koně}
    Start1, Start2: Index;  {souřadnice výchozí pozice koně}
    Nalezeno: boolean;      {příznak nalezení cesty koněm}
    i,j: Index;             {pomocné indexy}

procedure Cesta(i1,i2: Index; var Nalezeno: boolean);
  {Procedura pro nalezení cesty koně z pozice (i1,i2) do
   konce. Začátek cesty je uložen v globálním poli S.
   Výstupní parametr Nalezeno udává, zda se cestu podařilo
   nalézt (po vyzkoušení všech možností).}

  var Smer: 0..PocetTahu;{směr pohybu koně-index v poli Tah}
      j1,j2: integer;  {nová pozice koně}
      Krok: integer;   {pořadové číslo prováděného tahu}
      Nalez: boolean;  {příznak úspěšného nalezení cesty}

  begin
  Krok := S[i1,i2] + 1;
  Smer := 0;
```



```
  Nalez := false;           {zatím cesta nenalezena}
  repeat          {zkoušíme postupně všechny směry tahu koně}
    Smer := Smer+1;         {nový zkoušený směr}
    j1 := i1 + Tah[Smer].d1;
    j2 := i2 + Tah[Smer].d2;          {nové souřadnice koně}
    if (j1>=1) and (j1<=N) and (j2>=1) and (j2<=N) then
                                      {je uvnitř šachovnice}
      if S[j1,j2] = -1 then        {pole dosud nenavštíveno}
        begin
        S[j1,j2] := Krok;
        if Krok = PocetKroku then
          Nalez := true              {řešení úlohy nalezeno!}
        else
          begin             {prodloužit cestu pomocí rekurze}
          Cesta(j1,j2,Nalez);
          if not Nalez then
            S[j1,j2] := -1  {neúspěšný pokus - nutný návrat}
          end
        end
  until Nalez or (Smer=PocetTahu);
                           {konec hledání, pokud jsme cestu
                            našli nebo když jsme vyzkoušeli
                            všech 8 směrů bez úspěchu}
  Nalezeno := Nalez         {vrací se výsledek hledání}
  end;  {procedure Cesta}

begin  {program}
{inicializace pole tahů koněm:}
Tah[1].d1 := 1; Tah[1].d2 := 2;
Tah[2].d1 := 2; Tah[2].d2 := 1;
Tah[3].d1 := 2; Tah[3].d2 := -1;
Tah[4].d1 := 1; Tah[4].d2 := -2;
Tah[5].d1 := -1; Tah[5].d2 := -2;
Tah[6].d1 := -2; Tah[6].d2 := -1;
Tah[7].d1 := -2; Tah[7].d2 := 1;
Tah[8].d1 := -1; Tah[8].d2 := 2;

{inicializace šachovnice:}
write('Počáteční pozice koně: ');
readln(Start1, Start2);
for i:=1 to N do
```

```
  for j:=1 to N do S[i,j] := -1;
S[Start1,Start2] := 0;

Cesta(Start1,Start2,Nalezeno);

{výsledek hledání cesty:}
writeln;
if Nalezeno then  {výpis nalezené cesty koněm po šachovnici}
  for i:=1 to N do
    begin
    for j:=1 to N do write(S[i,j]:4);
    writeln
    end
else
  writeln('Cesta neexistuje.');
writeln
end.
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 3 | 56 | 19 | 40 | 5 | 42 | 21 |
| 33 | 18 | 1 | 4 | 57 | 20 | 39 | 6 |
| 2 | 55 | 34 | 59 | 36 | 41 | 22 | 43 |
| 17 | 32 | 47 | 52 | 45 | 58 | 7 | 38 |
| 48 | 13 | 54 | 35 | 60 | 37 | 44 | 23 |
| 31 | 16 | 51 | 46 | 53 | 26 | 61 | 8 |
| 12 | 49 | 14 | 29 | 10 | 63 | 24 | 27 |
| 15 | 30 | 11 | 50 | 25 | 28 | 9 | 62 |

*Obr. 18* Jedno z možných řešení úlohy pro výchozí pole v levém horním rohu šachovnice

Při řešení druhé úlohy, tj. nalezení nejkratší cesty koněm na šachovnici mezi dvěma danými poli, použijeme průchod do šířky. Do pole $S$ si budeme ukládat informace, jakým nejmenším počtem tahů se můžeme dostat na jednotlivá pole šachovnice. Výchozí pole obsahuje nulu. V první etapě výpočtu vyzkoušíme všechny možné tahy koně z výchozího pole



a všem polím přímo dostupným jedním tahem přiřadíme jedničku. Ve druhé etapě zkoušíme postupně všechny možné tahy koně ze všech polí s jedničkou. Uvažujeme ovšem pouze takové tahy, které vedou na dosud nenavštívená pole šachovnice (ta poznáme podle toho, že ještě obsahují hodnotu $-1$). Na tato pole zapíšeme dvojku. V další etapě budeme stejným způsobem zkoumat všechna pole s dvojkou a zapisovat trojky atd. Výpočet pokračuje tak dlouho, dokud šířící se „vlna" již navštívených polí nezasáhne pole cílové. V tom okamžiku známe délku nejkratší cesty mezi danými dvěma poli, je uložena v $S$ na pozici cílového pole.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | | 2 | 1 | 2 | | |
| 2 | | 0 | | 2 | | 2 | |
| 1 | 2 | | 2 | 1 | 2 | | |
| | 1 | 2 | 1 | | | 2 | |
| | 2 | | 2 | | 2 | | |
| 2 | | 2 | | 2 | | | |
| | | | | | | | |
| | | | | | | | |

*Obr. 19* Příklad šíření vlny na šachovnici

K úplnému dokončení celé úlohy nám ještě zbývá dořešit dva dílčí problémy. Prvním z nich je vhodná evidence nebo vyhledání všech polí na šachovnici obsahujících určitou hodnotu $K$. Tato pole potřebujeme znát k realizaci výpočetní etapy $K+1$. Druhou dosud otevřenou otázkou je způsob určení vlastní minimální cesty, neboť výše popsaný algoritmus určuje pouze její délku. Oba tyto úkoly můžeme řešit různými způsoby. Ukážeme si vždy dvě možná řešení. Jedno z nich bude jednodušší a paměťově úspornější, druhé bude zase rychlejší.

Vyhledat na šachovnici všechna pole s hodnotou $K$ můžeme jednoduše tak, že celé pole $S$ projdeme a otestujeme každý jeho prvek. Takové řešení se velmi snadno naprogramuje, nepotřebuje žádné další datové struktury a pro malé velikosti šachovnice není ani příliš pomalé. Druhou možností je řídit průchod šachovnicí pomocí fronty, podobně jako u procházení binárního stromu do šířky v kap. 8.1. Do pomocné fronty si budeme ukládat souřadnice všech polí, do kterých jsme již dorazili, ale z nichž jsme ještě nepokračovali v cestě dál. Vyhneme se

tím opakovanému prohledávání celé šachovnice, na začátku etapy $K+1$ máme souřadnice všech polí s hodnotou $K$ přímo uloženy ve frontě. Toto druhé řešení je paměťově trochu náročnější, ale zato rychlejší.

Po ukončení šíření „vlny" známe délku nejkratší cesty na cílové pole. Musíme ale ještě nalézt, kudy tato cesta vede. Cestu vyhledáváme tzv. zpětným průchodem, v obráceném směru od cílového pole k výchozímu. Nejjednodušší řešení opět vystačí s informacemi uloženými v poli $S$. Mezi přímými sousedy (tj. „sousedy" ve smyslu pohybu koně) cílového pole musí existovat alespoň jedno pole s hodnotou o 1 menší. Je-li takových polí více, znamená to, že existuje více různých cest minimální délky. My hledáme jednu jakoukoli nejkratší cestu, takže z těchto polí můžeme zvolit jedno libovolné a přesunout se do něj. Dále postupujeme stejným způsobem tak dlouho, až dorazíme na výchozí pole cesty. Tím je celá cesta určena. Zpětný průchod můžeme urychlit, jestliže si již během šíření vlny budeme zaznamenávat další vhodné informace. Použijeme pomocné pole, v němž si budeme ke každé pozici na šachovnici zaznamenávat jejího předchůdce na nějaké nejkratší cestě z výchozí pozice. Kdykoli vstoupíme poprvé na některé pole šachovnice, uložíme si nejen údaj, ve které etapě šíření vlny k tomu došlo, ale zapíšeme si k němu také souřadnice toho pole, odkud jsme právě přišli (to je jeho předchůdce na dosud nejkratší cestě). Zpětný průchod je pak velmi jednoduchý, od cílového vrcholu k výchozímu postupujeme přímo po zaznamenaných předchůdcích.

Celé řešení si nyní ukážeme naprogramované v Pascalu. Předvedeme si řešení úlohy hned ve dvou podobách. První program je napsán co nejjednodušším způsobem, bez jakýchkoli optimalizací rychlosti výpočtu. Nepoužívá frontu k řízení algoritmu vlny ani evidenci předchůdců pro zrychlení zpětného průchodu. Druhý program je naopak navržen tak, aby byl co nejrychlejší i při použití velké šachovnice. K řízení výpočtu je v něm využívána pomocná fronta realizovaná jednoduše v poli *Fronta* bez posouvání uložených záznamů. Víme totiž, že se ve frontě postupně vystřídá nejvýše $N^2$ polí (každé pole nejvýše jednou). V dalším pomocném poli *Pred* jsou zároveň zaznamenávány souřadnice předchůdců jednotlivých polí. Toto řešení má zaručenou časovou složitost $O(N^2)$, neboť postupně je navštíveno nejvýše $N^2$ polí na šachovnici a návštěva každého z nich představuje konstantní počet elementárních operací. Oba programy pro jednoduchost nekontrolují korektnost vstupních dat. Využívají také skutečnosti, že mezi libovolnými dvěma poli na čtvercové šachovnici velikosti



alespoň $4\times4$ existuje cesta šachovým koněm (netestují případ, že by cesta neexistovala).

```
program NejkratsiCestaKonem1;
{Nalezení nejkratší cesty koněm po šachovnici z daného
výchozího na cílové pole.
Použitá metoda řešení: prohledávání do šířky.
Šíření vlny řízeno pouze hodnotami na šachovnici.
Nalezení cesty zpětným průchodem podle hodnot pole S.}

const N = 8;          {velikost strany čtvercové šachovnice}
      PocetTahu = 8;  {počet různých tahů z jednoho pole}
type  Index = 1..N;   {typ indexu na šachovnici}

var S: array[Index,Index] of integer;              {šachovnice}
    Tah: array[1..PocetTahu] of
           record d1,d2:integer end;          {možné tahy koně}
    Start1, Start2: Index; {souřadnice výchozí pozice koně}
    Cil1, Cil2: Index;     {souřadnice cílové pozice koně}
    Krok: integer;         {pořadové číslo prováděného tahu}
    Smer: 0..PocetTahu;    {směr pohybu koně - index v Tah}
    i1,i2: Index;          {pozice koně}
    j1,j2: integer;        {nová pozice koně z (i1,i2)}

begin {program}
{inicializace pole tahů koněm:}
Tah[1].d1 := 1; Tah[1].d2 := 2;
Tah[2].d1 := 2; Tah[2].d2 := 1;
Tah[3].d1 := 2; Tah[3].d2 := -1;
Tah[4].d1 := 1; Tah[4].d2 := -2;
Tah[5].d1 := -1; Tah[5].d2 := -2;
Tah[6].d1 := -2; Tah[6].d2 := -1;
Tah[7].d1 := -2; Tah[7].d2 := 1;
Tah[8].d1 := -1; Tah[8].d2 := 2;

{inicializace šachovnice:}
write('Počáteční pozice koně: ');
readln(Start1, Start2);
write('Cílová pozice koně: ');
readln(Cil1, Cil2);
for i1:=1 to N do
```

```
  for i2:=1 to N do S[i1,i2] := -1;
 S[Start1,Start2] := 0;

 {algoritmus vlny:}
 Krok := 1;
 while S[Cil1,Cil2] = -1 do      {nemáme cestu na cílové pole}
  begin
  for i1:=1 to N do
    for i2:=1 to N do             {zkusíme jít z pole (i1,i2)}
      if S[i1,i2] = Krok-1 then
        for Smer:= 1 to PocetTahu do
                                  {zkusíme pohyb všemi směry}
          begin
          j1 := i1 + Tah[Smer].d1;
          j2 := i2 + Tah[Smer].d2;    {nové souřadnice koně}
          if (j1>=1) and (j1<=N) and
             (j2>=1) and (j2<=N) then {je uvnitř šachovnice}
            if S[j1,j2] = -1 then  {pole dosud nenavštíveno}
              S[j1,j2] := Krok     {na (j1,j2) dorazila vlna}
          end;
  Krok := Krok+1
  end;

{nalezení cesty zpětným průchodem podle hodnot pole S:}
writeln('Nejkratší cesta v obráceném pořadí:');
write('(', Cil1, ',', Cil2, ') ');
i1 := Cil1;
i2 := Cil2;
while (i1 <> Start1) or (i2 <> Start2) do
  begin
  Smer := 1;
  repeat
    j1 := i1 + Tah[Smer].d1;
    j2 := i2 + Tah[Smer].d2;    {nové souřadnice koně}
    if (j1>=1) and (j1<=N) and (j2>=1) and (j2<=N) then
                                  {je uvnitř šachovnice}
      if S[j1,j2] = S[i1,i2]-1 then
                {pole (i1,i2) je přímo dostupné z (j1,j2)}
        begin
        i1 := j1;
        i2 := j2;
```



```
        write('(', i1, ',', i2, ') ');
        Smer := 0          {už není třeba jít žádným směrem}
        end
      else
        Smer := Smer+1     {zkusíme jiný směr pohybu}
    else
      Smer := Smer+1       {zkusíme jiný směr pohybu}
  until Smer = 0           {už není třeba jít žádným směrem}
  end;
writeln
end.
```

---

```
program NejkratsiCestaKonem2;
{Nalezení nejkratší cesty koněm po šachovnici z daného
výchozího na cílové pole.
Použitá metoda řešení: prohledávání do šířky.
Šíření vlny řízeno pomocí fronty souřadnic polí.
Výpis cesty zpětným průchodem podle zaznamenaných hodnot
souřadnic předchůdců.}

const N = 8;           {velikost strany čtvercové šachovnice}
      PocetTahu = 8;   {počet různých tahů z jednoho pole}
      PocetPoli = 64;  { = N*N }
type  Index = 1..N;    {typ indexu na šachovnici}
      Souradnice = record s1,s2: Index end;
                       {souřadnice pole na šachovnici}

var S: array[Index,Index] of integer;          {šachovnice}
    Tah: array[1..PocetTahu] of
           record d1,d2: integer end;      {možné tahy koně}
    Pred: array[Index,Index] of Souradnice;    {předchůdci}
    Fronta: array[1..PocetPoli] of Souradnice;
                       {fronta pro řízení průchodu do šířky}
    ZacFr, KonFr: integer;          {začátek a konec fronty}
    Start1, Start2: Index; {souřadnice výchozí pozice koně}
    Cil1, Cil2: Index;     {souřadnice cílové  pozice koně}
    Krok: integer;         {pořadové číslo prováděného tahu}
    Smer: 0..PocetTahu;    {směr pohybu koně - index v Tah}
    i1,i2: Index;          {pozice koně}
    j1,j2: integer;        {nová pozice koně z (i1,i2)}
```

```
begin  {program}
{inicializace pole tahů koněm:}
Tah[1].d1 := 1; Tah[1].d2 := 2;
Tah[2].d1 := 2; Tah[2].d2 := 1;
Tah[3].d1 := 2; Tah[3].d2 := -1;
Tah[4].d1 := 1; Tah[4].d2 := -2;
Tah[5].d1 := -1; Tah[5].d2 := -2;
Tah[6].d1 := -2; Tah[6].d2 := -1;
Tah[7].d1 := -2; Tah[7].d2 := 1;
Tah[8].d1 := -1; Tah[8].d2 := 2;

{inicializace šachovnice a fronty:}
write('Počáteční pozice koně: ');
readln(Start1, Start2);
write('Cílová pozice koně: ');
readln(Cil1, Cil2);
for i1:=1 to N do
  for i2:=1 to N do S[i1,i2] := -1;
S[Start1,Start2] := 0;                    {výchozí pozice}
ZacFr := 1;
KonFr := 1;
Fronta[1].s1 := Start1;
Fronta[1].s2 := Start2;

{algoritmus vlny:}
while S[Cil1,Cil2] = -1 do      {nemáme cestu na cílové pole}
  begin
  i1 := Fronta[ZacFr].s1;
  i2 := Fronta[ZacFr].s2;       {zkusíme jít z pole (i1,i2)}
  ZacFr := ZacFr+1;                 {odebereme ho z fronty}
  Krok := S[i1,i2]+1;           {prováděný krok šíření vlny}
  for Smer:= 1 to PocetTahu do
                                 {zkusíme pohyb všemi směry}
    begin
    j1 := i1 + Tah[Smer].d1;
    j2 := i2 + Tah[Smer].d2;          {nové souřadnice koně}
    if (j1>=1) and (j1<=N) and
       (j2>=1) and (j2<=N) then       {je uvnitř šachovnice}
      if S[j1,j2] = -1 then         {pole dosud nenavštíveno}
        begin
        S[j1,j2] := Krok;           {na (j1,j2) dorazila vlna}
```



```
        KonFr := KonFr+1;               {vložíme ho do fronty}
        Fronta[KonFr].s1 := j1;
        Fronta[KonFr].s2 := j2;
        Pred[j1,j2].s1 := i1;  {jeho předchůdcem je (i1,i2)}
        Pred[j1,j2].s2 := i2
        end
    end;
  Krok := Krok+1
  end;

{výpis cesty zpětným průchodem podle hodnot Pred:}
writeln('Nejkratší cesta v obráceném pořadí:');
write('(', Cil1, ',', Cil2, ') ');
i1 := Cil1;
i2 := Cil2;
while (i1 <> Start1) or (i2 <> Start2) do
  begin
  j1 := Pred[i1,i2].s1;
  j2 := Pred[i1,i2].s2;               {souřadnice předchůdce}
  i1 := j1;
  i2 := j2;
  write('(', i1, ',', i2, ') ');
  end;
writeln
end.
```

### 8.3 Ořezávání a heuristiky

Algoritmy založené na metodě backtrackingu bývají velice pomalé. Zpravidla vedou k řešení úlohy s exponenciální časovou složitostí. Používáme je proto pouze tehdy, nedokážeme-li úlohu vyřešit jiným způsobem. Navíc se je snažíme alespoň v rámci možností co nejvíce zrychlit, i když tím jejich exponenciální složitost neodstraníme. Pro zlepšení algoritmů prohledávání do hloubky se používají dvě rozdílné techniky, a to ořezávání neperspektivních větví a volba pořadí větví podle heuristik. Možnosti jejich použití jsou velmi závislé na konkrétní úloze, zda se nám podaří najít nějaké kritérium pro ořezávání nebo vhodnou heuristiku.

**Ořezávání** použijeme tehdy, pokud jsme schopni na základě vyhodnocení momentálního stavu zjistit, že je to stav neperspektivní a že rozhodně

nepovede k řešení úlohy. V takovém případě by bylo zbytečné procházet strom všech možných cest výpočtu celý. Podstromy momentálního stavu nebudeme vůbec prohledávat, „odřízneme" je ze stromu a budeme se ihned vracet zpět, jako by byl momentální stav neúspěšným listem. Tato úprava algoritmu je korektní a neovlivní konečné řešení úlohy, pouze ho zrychlí tím, že se vynechá zkoumání neperspektivních variant výpočtu. Princip ořezávání si předvedeme na dvou příkladech.

Magickým čtvercem řádu $N$ rozumíme čtvercové schéma čísel velikosti $N \times N$, které obsahuje právě jednou každé celé číslo od 1 do $N^2$. Přitom platí, že součet čísel ve všech řádcích a ve všech sloupcích magického čtverce je stejný. Například pro $N = 3$ vypadá jeden z magických čtverců takto:

| | | |
|---|---|---|
| 2 | 9 | 4 |
| 7 | 5 | 3 |
| 6 | 1 | 8 |

Součet čísel v každém jeho řádku i v každém sloupci je 15. Naším úkolem nyní bude sestavit algoritmus, který najde a vytiskne všechny magické čtverce daného řádu $N$, pro hodnoty $N$ z předem známého rozmezí, např. od 3 do 8. Řešení založené na mechanickém backtrackingu bez jakéhokoli zlepšení bude zkoušet všechna rozmístění čísel od 1 do $N^2$ do čtverce velikosti $N \times N$. Postupně bude generovat všechna taková rozmístění a každé z nich zkoumat, zda se v něm shodují řádkové a sloupcové součty. Pravděpodobně bude postupovat systematicky, takže např. pro $N = 3$ nejprve umístí do první řady čtverce čísla 1, 2 a 3 a bude zkoušet všechna rozložení šesti zbývajících čísel, až neuspěje, vyzkouší první řadu 1, 2 a 4, potom 1, 2 a 5 atd. Takto vykoná zcela zbytečně obrovské množství výpočtů bez naděje na úspěch. Přitom stačí provést jednoduchou matematickou úvahu a spočítat si předem, čemu se součty čísel v řádcích a sloupcích magického čtverce řádu $N$ rovnají. Součet všech čísel od 1 do $N^2$ spočítáme jako aritmetický průměr z prvního a posledního čísla násobený počtem všech čísel. Je roven hodnotě $\frac{1}{2}N^2(N^2+1)$. Tato hodnota se rozdělí rovnoměrně mezi $N$ řádků se stejným součtem, takže každý řádek bude mít součet $\frac{1}{2}N(N^2+1)$. Až budeme zkoušet všechna rozmístění čísel od 1 do $N^2$ do čtverce, obsadíme nejprve jeho první řadu a hned zkontrolujeme její součet. Pokud se nerovná stanovené hodnotě, bylo by naprosto zbytečné zkoumat všechna možná rozmístění zbývajících čísel ve zbývajících řadách. Vrátíme se raději ihned k jinému obsazení první



řady čtverce. Obdobně postupujeme i po obsazení ostatních řad. V našem příkladě s $N = 3$ vychází řádkový součet 15, takže první řada tvaru 1, 2, 3 nebo 1, 2, 4 apod. je rozhodně neperspektivní. Detailnější řešení úlohy ponecháme již do cvičení.

Druhá úloha čerpá tematicky z problematiky grafů, které se budeme věnovat více v kap. 9. Mezi $N$ městy vede několik silnic. Každá silnice spojuje nějakou dvojici měst, mimo města se silnice nijak nespojují ani nekříží. Každé město je označeno názvem. Na každém konci každé silnice je uvedeno, kam vede a jakou má délku. Stojíte ve městě $K$ a máte nalézt nejkratší cestu do města $L$. Neznáte ovšem celkový plán všech existujících silnic a jejich délek. K dispozici máte vždy jen informace o silnicích vedoucích z toho města, kde momentálně jste. Můžete si samozřejmě dělat jakékoli poznámky. Vzhledem k podmínkám úlohy není možné použít pro určení nejkratší cesty v grafu žádný standardní grafový algoritmus (srov. Dijkstrův algoritmus v kap. 9.4). Nezbývá nám než použít průchod sítí silnic do hloubky. Procházení do šířky také nepřipadá v úvahu, a to hned z několika důvodů. Předně jeden člověk nemůže souběžně zkoumat více cest najednou. I kdybyste se ale mohli „rozdvojit", stejně by nám průchod do šířky žádné vylepšení nepřinesl. Máme sice nalézt nejkratší cestu z $K$ do $L$, nikoli však nejkratší co do počtu navštívených měst, ale z hlediska součtu délek prošlých silnic. Tato dvě kritéria mohou vést k odlišnému výsledku, takže i v případě průchodu do šířky bychom stejně museli vyzkoušet všechny cesty a z jejich délek vzít minimum.

Při procházení si v každém městě vždy nějak uspořádáme silnice, které z něj vedou (třeba abecedně podle názvů cílových měst). Postupně je pak budeme zkoumat. Přitom budeme průběžně počítat vzdálenost od výchozího města $K$ a dáme pozor, abychom do některého města nepřišli opakovaně (neboť nejkratší cesta z $K$ do $L$ jistě neobsahuje smyčku). Kdykoli během procházení přijdeme do cílového města $L$, zaznamenáme si dosaženou vzdálenost z města $K$ a použitou cestu a pak se začneme vracet a zkoumat další možné cesty. Právě popsaným způsobem by pracoval základní algoritmus založený na jednoduchém backtrackingu. Nyní si ho trochu vylepšíme a zrychlíme pomocí ořezávání. Můžeme použít hned dva typy ořezávání najednou, z nichž jedno bychom mohli nazvat „lokální" (rozhoduje se v něm na základě informací týkajících se právě navštíveného města) a druhé „globální" (rozhodujeme na základě znalosti pocházející odjinud, podle dosud nejlepšího nalezeného řešení úlohy).

K realizaci ořezávání bude potřeba evidovat si během procházení, která města jsme již někdy dříve navštívili a jakou dosud nejkratší cestou z města $K$ jsme do nich došli. Kdykoli během procházení vstoupíme do nějakého města $M$, kde jsme byli již někdy dříve při zkoumání jiné cesty, porovnáme aktuální vzdálenost z města $K$ do města $M$ (tj. vzdálenost měřenou po právě procházené cestě) se zaznamenanou dosud nejmenší dříve nalezenou vzdáleností z $K$ do $M$. Pokud není aktuální vzdálenost menší, nemá smysl pokračovat v prohledávání z města $M$ dál. Ve městě $M$ jsme totiž již byli a cesty vedoucí z $M$ jsme již někdy dříve procházeli, a to s příznivější (popř. stejnou) „vstupní vzdáleností" do $M$.

Druhé, globální ořezávání se uplatní až od chvíle, kdy poprvé vstoupíme do cílového města $L$. Kdykoli později prodloužíme právě zkoumanou cestu, porovnáme její momentální délku s délkou dosud nejkratší nalezené cesty z $K$ do $L$. Jestliže není právě zkoumaná cesta kratší, je neperspektivní a nemá cenu dále ji prodlužovat. Všechny vzdálenosti měst jsou totiž kladné, takže tato cesta již nemůže v žádném případě vylepšit výslednou hodnotu.

Zcela odlišnou metodou zrychlování výpočtu je použití vhodné **heuristiky**. Přesná definice pojmu heuristika je dosti obtížná, pokusíme se proto raději o trochu širší slovní vysvětlení. Heuristika je návod, který nám říká, jaký postup řešení úlohy vede obvykle k rychlému dosažení výsledku. Není to tedy pravidlo, které by zrychlení výpočtu zaručovalo, pouze nám s velkou pravděpodobností pomůže, a proto je dobré se jím řídit. Odvodit dobrou účinnou heuristiku pro řešení dané úlohy nebývá snadné, je třeba vyjít z hlubší znalosti řešeného problému a ze zkušenosti, jak bývá dobré při řešení podobných úloh postupovat. Získání použitelné heuristiky je často více věcí odhadu a intuice řešitele než výsledkem nějakého exaktního odvození.

Zopakujme si ještě jednou, že heuristika nám nic nezaručuje, ale samozřejmě nesmí ani nic pokazit. Uplatnění dobré heuristiky v algoritmu řešení úlohy by mělo pro většinu vstupních dat vést k relativně rychlému získání výsledku, není však vyloučeno, že pro některá vstupní data nám heuristika nepomůže a výpočet zůstane pomalý. Při řešení úloh založených na metodě prohledávání do hloubky využíváme heuristiky ke stanovení pořadí, v jakém budeme zkoumat možné cesty vedoucí z nějakého stavu. Nejdříve bychom chtěli procházet ty cesty, na nichž očekáváme rychlejší nalezení řešení úlohy. Správnost backtrackingu nezávisí



na pořadí, v jakém jednotlivé cesty procházíme, nesmíme jenom žádnou vynechat, dokud nenajdeme řešení. Toto pořadí se obvykle volí nějakým jednoduchým pevně daným způsobem. Například v naší úloze z kap. 8.2 o koni na šachovnici bylo pořadí zkoumaných cest pevně určeno naplněním pole *Tah*. Rozhodně tedy nic nepokazíme, když takovéto implicitní pořadí nějak pozměníme, máme-li k tomu důvod, byť založený jen na intuitivně odvozené heuristice. Změna pořadí tahů může ovlivnit pouze rychlost výpočtu. Má vliv také na to, které řešení nalezneme jako první, pokud má úloha více různých řešení. Jestliže byla použitá heuristika dobrá, najdeme řešení úlohy rychleji, jestliže však dobrá nebyla, o řešení nepřijdeme (v nejhorším případě nešťastně zvolené „heuristiky" pouze výpočet zpomalíme a prodloužíme).

Pokud pro nějaká vstupní data úloha nemá řešení, žádná heuristika nám pochopitelně nepomůže, negativní výsledek získáme až po prozkoumání všech cest. Heuristika nám nepomůže také tehdy, jestliže chceme najít všechna řešení dané úlohy. I v tomto případě musíme projít celý strom všech možných cest výpočtu a volbou pořadí nic neušetříme. Naopak, uplatnění heuristiky pro stanovení pořadí zkoumaných cest představuje samo o sobě určitou práci navíc, takže výpočet se dokonce trochu zpomalí.

Použití heuristiky si ukážeme na úloze projít šachovým koněm celou šachovnici velikosti $N \times N$. Tuto úlohu jsme již vyřešili v kap. 8.2 pomocí průchodu do hloubky. Řešení z kap. 8.2 nyní upravíme přidáním heuristiky pro určení pořadí, v němž zkoumáme možné tahy koněm. V původním řešení úlohy bylo toto pořadí pevně určeno zaplněním konstantního pole *Tah*. Zkusme nyní použít následující heuristiku: Má-li kůň za sebou nějakou část cesty po šachovnici a zkoumáme-li další možné tahy z momentálně posledního pole jeho cesty, půjdeme nejprve na ta dosud nenavštívená pole, z nichž bude nejméně možností dalšího bezprostředního pokračování cesty. Všechny přípustné tahy koně tedy uspořádáme vždy podle toho, kolik možností tahu budeme mít v následujícím kroku, a to od nejmenšího počtu k největšímu. V tomto pořadí je budeme zkoušet.

Modifikaci řešení z kap. 8.2 s uvedenou heuristikou si předvedeme také v Pascalu. Porovnejte si oba programy, že se skutečně liší pouze přidáním této heuristiky pro určení pořadí tahů. Heuristika je to sice celkem jednoduchá, ale délka našeho programu se tím přesto zdvojnásobila.

```
program Heuristika;
{Nalezení takové cesty koněm po šachovnici, aby bylo každé
pole šachovnice navštíveno právě jednou.
Použitá metoda řešení: prohledávání s návratem, realizované
pomocí rekurzivní procedury.
Pořadí zkoušených směrů pohybu koně řízeno heuristikou:
nejprve se jde na pole s menším počtem možností dalšího
pokračování cesty.}

const N = 8;            {velikost strany čtvercové šachovnice}
      PocetKroku = 63;  { = N*N-1 }
                {počet tahů koně tvořících výslednou cestu}
      PocetTahu = 8;  {počet různých tahů z jednoho pole}
type  Index = 1..N;    {typ indexu na šachovnici}
      SeznamSmeru = array[1..PocetTahu] of 1..PocetTahu;
                    {seznam přípustných směrů pohybu koně}

var S: array[Index,Index] of integer;            {šachovnice}
    Tah: array[1..PocetTahu] of
           record d1,d2:integer end;        {možné tahy koně}
    Start1, Start2: Index;  {souřadnice výchozí pozice koně}
    Nalezeno: boolean;      {příznak nalezení cesty koněm}
    i,j: Index;             {pomocné indexy}

function PocetVolnych(i1,i2: Index): integer;
  {Pomocná funkce pro určení počtu volných polí v okolí
   pole (i1,i2) - výsledek vrací jako funkční hodnotu.}

  var Smer: 1..PocetTahu;{směr pohybu koně-index v poli Tah}
      j1,j2: integer;  {nová pozice koně}
      Pocet: integer;  {počet volných polí v okolí (i1,i2)}

  begin
  Pocet := 0;
  for Smer:=1 to PocetTahu do
    begin    {zkoušíme postupně všechny směry tahu koně}
    j1 := i1 + Tah[Smer].d1;
    j2 := i2 + Tah[Smer].d2;          {nové souřadnice koně}
    if (j1>=1) and (j1<=N) and (j2>=1) and (j2<=N) then
```



```
                                      {je uvnitř šachovnice}
      if S[j1,j2] = -1 then         {pole dosud nenavštíveno}
        Pocet := Pocet+1;
    end;
  PocetVolnych := Pocet
  end;  {function PocetVolnych}

procedure UrciSmery(i1,i2: Index; var Sm: SeznamSmeru;
                    var PocetSm: integer);
  {Pomocná procedura pro nalezení všech možných tahů koně
   z pozice (i1,i2). Směry tahů uloží do pole Sm, jejich
   počet do PocetSm. Tahy seřadí podle heuristiky.}

  var Smer: 1..PocetTahu;{směr pohybu koně-index v poli Tah}
      j1,j2: integer;  {nová pozice koně}
      Volne: array[1..PocetTahu] of 0..PocetTahu;
        {počet volných polí v okolí pole určeného podle Sm}
      i,j,k,x: integer;          {pro potřeby třídění}

  begin
  PocetSm := 0;
  for Smer:=1 to PocetTahu do
    begin        {zkoušíme postupně všechny směry tahu koně}
    j1 := i1 + Tah[Smer].d1;
    j2 := i2 + Tah[Smer].d2;          {nové souřadnice koně}
    if (j1>=1) and (j1<=N) and (j2>=1) and (j2<=N) then
                                      {je uvnitř šachovnice}
      if S[j1,j2] = -1 then         {pole dosud nenavštíveno}
        begin
        PocetSm := PocetSm+1;
        Sm[PocetSm] := Smer;     {uložit úspěšný směr pohybu}
        Volne[PocetSm] := PocetVolnych(j1,j2)
        end
    end;
  {pole Sm je naplněno přípustnými tahy, ještě musíme
  zvolit jejich pořadí - heuristika podle počtu volných
  sousedních polí:}
  for i:=1 to PocetSm-1 do
    begin
    k:=i;
    for j:=i+1 to PocetSm do
```

```
    if Volne[j] < Volne[k] then k:=j;
    if k > i then
      begin x:=Sm[k]; Sm[k]:=Sm[i]; Sm[i]:=x;
            x:=Volne[k]; Volne[k]:=Volne[i]; Volne[i]:=x
      end
    end
  end;  {procedure UrciSmery}

procedure Cesta(i1,i2: Index; var Nalezeno: boolean);
  {Procedura pro nalezení cesty koně z pozice (i1,i2) do
   konce. Začátek cesty je uložen v globálním poli S.
   Výstupní parametr Nalezeno udává, zda se cestu podařilo
   nalézt (po vyzkoušení všech možností)}

  var Smer: 1..PocetTahu;{směr pohybu koně-index v poli Sm}
      Sm: SeznamSmeru;
        {možné směry pohybu koně v pořadí daném heuristikou
        - indexy do pole Tah}
      PocetSm: integer;   {počet možných tahů koně}
      j1,j2: integer;  {nová pozice koně}
      Krok: integer;   {pořadové číslo prováděného tahu}
      Nalez: boolean;  {příznak úspěšného nalezení cesty}

  begin
  Krok := S[i1,i2] + 1;
  UrciSmery(i1,i2,Sm,PocetSm);
  Smer := 1;
  Nalez := false;
  while (Smer <= PocetSm) and not Nalez do
    begin       {zkoušíme postupně všechny směry tahu koně}
    j1 := i1 + Tah[Sm[Smer]].d1;
    j2 := i2 + Tah[Sm[Smer]].d2;      {nové souřadnice koně}
    S[j1,j2] := Krok;
    if Krok = PocetKroku then
      Nalez := true                   {řešení úlohy nalezeno!}
    else
      begin                {prodloužit cestu pomocí rekurze}
      Cesta(j1,j2,Nalez);
      if not Nalez then
        S[j1,j2] := -1      {neúspěšný pokus - nutný návrat}
      end;
```



```
    Smer := Smer+1;          {nový zkoušený směr}
    end;
  Nalezeno := Nalez          {vrací se výsledek hledání}
  end;  {procedure Cesta}

begin  {program}
{inicializace pole tahů koněm:}
Tah[1].d1 := 1; Tah[1].d2 := 2;
Tah[2].d1 := 2; Tah[2].d2 := 1;
Tah[3].d1 := 2; Tah[3].d2 := -1;
Tah[4].d1 := 1; Tah[4].d2 := -2;
Tah[5].d1 := -1; Tah[5].d2 := -2;
Tah[6].d1 := -2; Tah[6].d2 := -1;
Tah[7].d1 := -2; Tah[7].d2 := 1;
Tah[8].d1 := -1; Tah[8].d2 := 2;

{inicializace šachovnice:}
write('Počáteční pozice koně: ');
readln(Start1, Start2);
for i:=1 to N do
  for j:=1 to N do S[i,j] := -1;
S[Start1,Start2] := 0;

Cesta(Start1,Start2,Nalezeno);
{výsledek hledání cesty:}
writeln;
if Nalezeno then  {výpis nalezené cesty koněm po šachovnici}
  for i:=1 to N do
    begin
    for j:=1 to N do write(S[i,j]:4);
    writeln
    end
else
  writeln('Cesta neexistuje.');
writeln
end.
```

Je velmi zajímavé vyzkoušet si na počítači rychlost výpočtu obou programů, původního z kap. 8.2 a tohoto s heuristikou. Zjistíte, že použitá heuristika je velmi dobrá a účinná. Rychlost výpočtu původního programu značně závisí na volbě výchozí pozice koně. Na běžném počítači

typu PC se v případě šachovnice $5 \times 5$ pohybuje doba výpočtu v rozmezí řádově od setin sekundy do stovek sekund, na šachovnici velikosti $8 \times 8$ však již výpočet trvá od několika minut po mnoho hodin. Naproti tomu při použití naší heuristiky délka výpočtu téměř nezávisí na určení výchozí pozice, na šachovnici $5 \times 5$ i $8 \times 8$ získáme řešení vždy za několik málo setin sekundy. V případě běžné šachovnice rozměrů $8 \times 8$ polí tedy heuristika přináší až neuvěřitelné zrychlení výpočtu, upravený program určí výslednou cestu koně přibližně stotisíckrát rychleji.

## CVIČENÍ

**1.** Navrhněte dynamickou datovou reprezentaci obecného stromu a naprogramujte procedury pro průchod tímto stromem do hloubky a do šířky.

**2.** Řešte první ukázkovou úlohu z kap. 8.2 — průchod šachovým koněm celou šachovnicí — bez použití rekurze.

**3.** Pozměníme zadání druhé ukázkové úlohy z kap. 8.2 tak, že máme nalézt jednu libovolnou (ne nutně nejkratší) cestu šachového koně mezi danými dvěma poli na šachovnici. Jak budete postupovat při řešení?

**4.** Na čtvercové šachovnici velikosti $N \times N$ nalezněte nejkratší cestu šachového krále z daného výchozího na cílové pole. Některá pole šachovnice jsou obsazena, na ně král nesmí vstoupit. Vstupem programu budou souřadnice výchozího, cílového a zakázaných polí. Pamatujte na to, že cesta krále z výchozího na cílové pole nemusí vůbec existovat, neboť obsazená pole mezi nimi mohou vytvořit neprostupnou hradbu.

**5.** Napište program řešící úlohu z kap. 8.3 vygenerovat všechny magické čtverce řádu $N$. Pokuste se co nejvíce urychlit výpočet.



# 9 PRÁCE S GRAFY

Grafy patří k důležitým matematickým strukturám, které se uplatňují při řešení mnoha praktických úloh. Základní pojmy týkající se grafů jsme si připomenuli v kap. 2.2. Je dobře známo a vyzkoušeno, jaké způsoby datové reprezentace grafu v programu jsou vhodné. Je také známa řada standardních algoritmů pro práci s grafy. Najdete je v každé základní učebnici teorie grafů nebo diskrétní matematiky (např. [2], [15]). V běžně dostupné literatuře však tyto algoritmy bývají uváděny většinou pouze teoreticky, bez návodu, jak je co nejšikovněji naprogramovat. My se zde proto omezíme jen na několik nejzákladnějších grafových algoritmů, ale zato si ukážeme i způsob jejich efektivní programové realizace. Vybereme přitom ty algoritmy, které se nejčastěji uplatňují při řešení úloh z praxe.

## 9.1 Reprezentace grafu v programu

Budeme-li řešit na počítači úlohu, v níž se budou provádět nějaké manipulace s grafy, budeme si muset především zvolit vhodnou datovou strukturu pro reprezentaci grafu v programu. Běžně se používá několik různých reprezentací a nedá se obecně říci, která je lepší a která horší. Záleží vždy na tom, co o grafu víme a co s ním budeme chtít provádět.

Ukážeme si nyní přehledně několik typických a nejpoužívanějších reprezentací grafu. V dalších odstavcích pak uvidíme vhodnou volbu reprezentace při řešení konkrétních grafových úloh. Budeme nadále předpokládat, že chceme uložit graf o $N$ vrcholech a $M$ hranách pro dané konstanty $N$, $M$. Dále předpokládáme, že vrcholy grafu jsou označeny čísly od 1 do $N$, popř. hrany čísly od 1 do $M$.

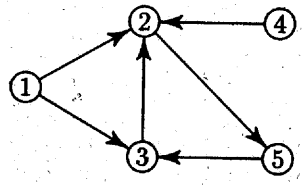
1. **Matice sousednosti** slouží k uchování neohodnocených grafů, neorientovaných i orientovaných. Jejím obsahem je informace, které dvojice vrcholů jsou spojeny hranou a které ne. Je to matice velikosti $N \times N$, jejími prvky jsou logické hodnoty `true/false` (nebo ve stejném významu čísla 1 a 0):

```
var Soused: array [1..N,1..N] of boolean;
```

Prvky matice přímo určují, mezi kterými dvojicemi vrcholů vede hrana:

$Soused[x,y]$ = `true` ... graf obsahuje hranu $(x,y)$,
$Soused[x,y]$ = `false` ... graf neobsahuje hranu $(x,y)$.

Jedná-li se o neorientovaný graf, je matice sousednosti symetrická, tzn. $Soused[x,y] = Soused[y,x]$ pro každou dvojici vrcholů $x$, $y$. V případě orientovaného grafu matice sousednosti symetrická být nemusí. Hodnota $Soused[x,y]$ potom udává, zda vede orientovaná hrana ve směru z vrcholu $x$ do vrcholu $y$.




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 |

*Obr. 20* Matice sousednosti orientovaného grafu

2. **Matice vzdáleností** je obdobou matice sousednosti pro případ ohodnocených grafů. Je to opět matice tvaru $N \times N$, do které se ukládají ohodnocení jednotlivých hran, popř. příznak, že hrana neexistuje. Pokud bychom tedy uvažovali ohodnocení hran grafu celými kladnými čísly, měla by deklarace matice vzdáleností třeba tento tvar:

```
var Vzdalenost: array [1..N,1..N] of integer;
```

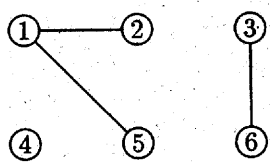
Význam uložených hodnot:

$Vzdalenost[x,y] > 0$ ... graf obsahuje hranu $(x,y)$ s ohodnocením $Vzdalenost[x,y]$
$Vzdalenost[x,y] = 0$ ... graf neobsahuje hranu $(x,y)$.

Stejně jako v případě matice sousednosti i zde platí, že pro neorientované grafy je matice vzdáleností symetrická, zatímco pro grafy orientované být symetrická nemusí.



3. **Matice incidence** je méně používaná reprezentace grafu, která slouží zejména pro případ grafů s mnoha vrcholy a málo hranami. Matice sousednosti nebo matice vzdáleností by zde byla velmi rozsáhlá a přitom by obsahovala skoro samé nuly. Matice incidence je matice tvaru $M \times N$, tj. každý řádek matice odpovídá jedné hraně a každý sloupec jednomu vrcholu. V každém řádku je přímo vyznačena dvojice vrcholů, mezi nimiž příslušná hrana vede. Jsou v něm tedy přesně dva nenulové údaje, zatímco všechny zbývající prvky na řádku mají hodnotu 0.




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 0 | 0 | 1 |

*Obr. 21* Matice incidence neorientovaného grafu

Matici incidence můžeme použít v různých obměnách pro případ grafů neorientovaných i orientovaných, neohodnocených i ohodnocených. Pro neorientované neohodnocené grafy postačí matice logických hodnot, v níž budou na každém řádku přesně dvě hodnoty `true` (ve sloupcích, mezi kterými vrcholy hrana vede):

```
var Incidence1: array [1..M,1..N] of boolean;
```

V případě orientovaného grafu již potřebujeme ukládat do matice tři různé hodnoty, a proto použijeme matici celých čísel. V každém řádku matice budou dvě nenulová čísla: jednou $-1$ pro označení vrcholu, odkud hrana vede, jednou $+1$ pro označení vrcholu, do kterého hrana vede:

```
var Incidence2: array [1..M,1..N] of -1..1;
```

Pro ohodnocené grafy musíme do matice ukládat také ohodnocení hran. Pokud bychom tedy uvažovali ohodnocení každé hrany grafu kladným celým číslem, stačí dát do matice místo hodnoty `true` (pro neorientované grafy) nebo $+1$ (pro orientované grafy) přímo kladné celé číslo udávající ohodnocení hrany:

```
var Incidence3: array [1..M,1..N] of integer;
```

4. **Seznam následníků** jednotlivých vrcholů představuje jednoduchou úspornou reprezentaci grafu, která je pro mnohé grafové algoritmy velmi výhodná. Tato reprezentace spočívá v tom, že si ke každému vrcholu grafu pamatujeme seznam čísel těch vrcholů, do kterých z něj vede hrana. Vytvářet seznam následníků jednotlivých vrcholů dynamicky by bylo trochu pracné a mít pro každý vrchol samostatné pole na uložení jeho následníků zase paměťově velmi neúsporné. Používají se zde proto s výhodou dvě pole: v poli $E$ o velikosti $M$ jsou umístěna čísla všech následníků, do pole $V$ o velikosti $N+1$ pak ukládáme indexy určující, kde jsou v poli $E$ uloženi následníci toho kterého vrcholu:

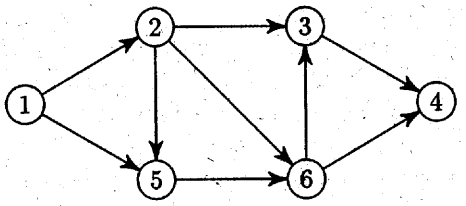
```
var V: array [1..N+1] of 1..M+1;
    E: array [1..M] of 1..N;
```

Hodnoty $N+1$ a $M+1$ v deklaraci pole $V$ jsou pouze z technických důvodů, aby se s reprezentací grafu lépe pracovalo (aby bylo možné jednotným způsobem procházet i následníky posledního vrcholu s číslem $N$). Vrchol grafu s číslem $x$ má své následníky uložen v poli $E$ počínaje prvkem s indexem $V[x]$. Protože od pozice $V[x+1]$ začínají v poli $E$ následníci vrcholu $x+1$, nachází se poslední následník vrcholu $x$ na pozici $V[x+1]-1$. Následníky vrcholu $x$ jsou tedy vrcholy s čísly $E[V[x]], E[V[x]+1], \ldots, E[V[x+1]-1]$. Jestliže nějaký vrchol $y$ nemá žádného následníka, položíme jednoduše $V[y] = V[y+1]$.

Reprezentace grafu seznamem následníků jednotlivých vrcholů se stejným způsobem používá pro grafy neorientované i orientované. Ve své základní podobě, kterou jsme si právě popsali, slouží pro grafy neohodnocené, lze ji však snadno modifikovat i pro ohodnocené grafy (viz cvičení 1.).

5. **Seznam hran** grafu je velmi primitivní reprezentace, která je každému srozumitelná, snadno se vytváří a je i paměťově dosti úsporná. Můžeme ji použít pro grafy orientované i neorientované, neohodnocené i ohodnocené. Její velkou nevýhodou však je, že se s ní dost špatně pracuje. V mnoha algoritmech totiž potřebujeme provádět takové akce, jako nalézt všechny hrany vedoucí z daného vrcholu. Podobné operace se ve všech ostatních uvedených reprezentacích grafu provádějí snáze






| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $V$ | 1 | 3 | 6 | 7 | 7 | 8 | 10 |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $E$ | 2 | 5 | 3 | 5 | 6 | 4 | 6 | 3 | 4 |

*Obr. 22* Příklad reprezentace grafu seznamem následníků

a rychleji. K uložení seznamu hran můžeme použít pole (známe-li jejich počet nebo alespoň rozumný horní odhad jejich počtu), popř. spojový seznam. Prvky pole nebo seznamu budou obsahovat dvě čísla vrcholů, mezi nimiž hrana vede, a v případě ohodnocených grafů ještě ohodnocení hrany.

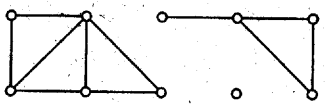
**Příklad**

Budeme reprezentovat orientovaný ohodnocený graf obsahující předem známý počet $N$ vrcholů a $M$ hran. Každá hrana má přiřazeno ohodnocení dvěma reálnými čísly:

```
var Graf: array [1..M] of record
                            Odkud, Kam: 1..N;
                            Hod1, Hod2: real
                          end;
```

## 9.2 Komponenty souvislosti grafu

Nejprve si vysvětlíme pojem **souvislost grafu**. Říkáme, že graf je souvislý, jestliže mezi libovolnou dvojicí jeho vrcholů existuje cesta. Název souvislost je velmi výstižný, graf je souvislý právě tehdy, pokud jsou všechny jeho vrcholy „pospojovány" a graf „drží pohromadě". Naším úkolem nyní bude zjistit, zda je daný graf souvislý. Pokud souvislý není, budeme chtít určit jeho **komponenty souvislosti**. To jsou co největší souvislé části grafu. Mezi libovolnými dvěma vrcholy z téže komponenty souvislosti existuje cesta, zatímco mezi dvěma vrcholy z různých komponent souvislosti cesta neexistuje.



*Obr. 23* Graf se třemi komponentami souvislosti

Při určování souvislosti grafu a hledání jeho komponent souvislosti se budeme zajímat pouze o neorientované grafy. Případné ohodnocení grafu s určováním souvislosti nijak nesouvisí. Budeme proto považovat všechny grafy za neohodnocené.

Úlohu můžeme přeformulovat i do řeči „silniční sítě". Je dán počet měst a seznam přímých silnic spojujících dvojice měst. Žádné silnice se nekříží mimo města (všechna případná křížení silnic mimo města jsou mimoúrovňová). Zjistěte, zda je možné dojet po silnicích z kteréhokoli města do kteréhokoli jiného města. Pokud ne, rozdělte všechna města do co největších skupin tak, aby v rámci každé skupiny existovalo silniční spojení mezi všemi dvojicemi měst.

Při testování souvislosti grafu a určování komponent souvislosti využijeme libovolný algoritmus na procházení daným grafem. Můžeme použít průchod do hloubky, do šířky nebo v jakémkoli jiném pořadí (viz kap. 8). Zvolíme nějaký výchozí vrchol, třeba vrchol s číslem 1. Z něj začneme procházet graf, přičemž postupně navštívíme a označíme všechny dostupné vrcholy. To znamená, že jsme označili všechny ty vrcholy, které leží v téže komponentě souvislosti jako výchozí vrchol číslo 1. Jsou-li nyní označeny všechny vrcholy grafu, je tento graf souvislý. V opačném případě jsme právě určili jednu jeho komponentu souvislosti. Tentýž postup pak uplatníme na zbytek grafu. Zvolíme tedy jeden libovolný



dosud nenavštívený vrchol (třeba ten s nejmenším číslem) a z něj budeme graf procházet. Dostupné vrcholy budeme opět označovat, tentokrát nějakou jinou značkou, abychom komponenty od sebe odlišili. Po skončení průchodu máme vyznačenu druhou komponentu souvislosti. Celý postup opakujeme tak dlouho, až budou označeny všechny vrcholy grafu. Různá označení vrcholů pak určují jednotlivé komponenty souvislosti daného grafu.

Popsaný algoritmus má kvadratickou časovou složitost vzhledem k počtu vrcholů grafu. Postupně musí být označeno všech $N$ vrcholů. Po označení každého z nich pak musíme zkoumat všechny jeho následníky, kterých může být až $N$. Celkově se tedy provede až $N^2$ elementárních operací. Pokud bychom chtěli získat přesnější odhad časové složitosti našeho algoritmu, můžeme použít k jejímu vyjádření více parametrů charakterizujících velikost konkrétní řešené úlohy. Graf je tvořen $N$ vrcholy a $M$ hranami. Při postupném zkoumání následníků všech označených vrcholů se každá hrana grafu použije dvakrát (jednou v každém směru), takže celkový počet provedených elementárních operací lze odhadnout výrazem $O(N + M)$.

Při výběru vhodné vnitřní reprezentace grafu vycházíme z toho, jaké operace s grafem budeme provádět. V tomto případě potřebujeme vždy k vrcholu nalézt všechny jeho sousedy, takže nejvhodnější bude uložení grafu pomocí seznamu následníků jednotlivých vrcholů. Použít bychom mohli i matici sousednosti. K provádění vlastního výpočtu je vhodné použít dvě pomocná pole. Jedno z nich bude indexováno čísly vrcholů a bude se do něj ukládat informace, do které komponenty souvislosti jednotlivé vrcholy patří, resp. příznak, že vrchol ještě nebyl navštíven. Druhé pole slouží k realizaci průchodu grafem. Ukládají se do něj čísla těch vrcholů právě zkoumané komponenty souvislosti, do kterých již dospělo prohledávání (byly označeny), ale z nichž ještě prohledávání nepokračovalo (ještě jsme nezkoumali jejich následníky). Prohledávání komponenty vždy pokračuje některým vrcholem z tohoto druhého pole. Pokud budeme brát vždy ten poslední a pracovat tedy s polem jako se zásobníkem, budeme provádět průchod grafem do hloubky. Jestliže z pole vezmeme vždy první vrchol jako v případě fronty, průchod grafem bude probíhat do šířky (bližší vysvětlení viz kap. 8.1). V principu by bylo možné zvolit i jakýkoli jiný postup výběru vrcholů k dalšímu zpracování, ale realizace takového výběru by byla jen zbytečně komplikovanější.

Popsaný postup si nyní ukážeme ještě ve tvaru programu. Pro zjednodušení těch částí programu, které nejsou podstatné z hlediska celkového principu řešení, budeme předpokládat, že na vstupu je graf zadán počtem vrcholů $N$ a pak přímo seznamy následníků jednotlivých vrcholů v pořadí od vrcholu číslo 1 k vrcholu $N$ (následníci jednoho vrcholu vždy na jednom řádku). Protože graf je neorientovaný, znamená to, že například při zadání hrany mezi vrcholy 2 a 5 musíme zapsat číslo 5 mezi sousedy vrcholu číslo 2 a číslo 2 mezi sousedy vrcholu číslo 5. Program dále pro jednoduchost nekontroluje korektnost vstupních dat (zda je počet vrcholů a hran grafu dostatečně malý s ohledem na definici konstant v programu, zda jsou čísla všech vrcholů z povoleného rozmezí a zda je zadaný graf opravdu neorientovaný). Při zkoumání komponent souvislosti budeme graf procházet do hloubky, neboť zásobník se ze všech typů seznamů nejsnáze programuje.

---

```
program Souvislost;
{Stanovení komponent souvislosti grafu}

const MaxVrch = 100;        {max. přípustný počet vrcholů}
      MaxVrchPlus1 = 101;   {MaxVrch+1}
      MaxHran = 1000;       {max. přípustný počet hran}
      MaxHranPlus1 = 1001;  {MaxHran+1}

var V: array[1..MaxVrchPlus1] of 1..MaxHranPlus1;
    E: array[1..MaxHran] of 1..MaxVrch;
         {uložení grafu ve tvaru seznamu následníků}
    Komponenta: array[1..MaxVrch] of integer;
         {pro každý vrchol číslo jeho komponenty}
    Zasobnik: array[1..MaxVrch] of 1..MaxVrch;
         {zásobník vrcholů pro další zpracování}
    Vrchol: 0..MaxVrch;     {vrchol zásobníku}
    Navstiven: 0..MaxVrch; {počet navštívených vrcholů}
    N: 1..MaxVrch;          {počet všech vrcholů grafu}
    Barva: integer;         {barva vytvářené komponenty}
    Zacatek: 1..MaxVrch;    {nejmenší vrchol komponenty}
    i,j: integer;

begin
```



```
{Načtení vstupních dat - viz text výše:}
write('Počet vrcholů grafu: ');
readln(N);
writeln('Po řádcích následníci jednotlivých vrcholů',
        ' od 1 do ', N, ':');
V[1]:=1;
j:=1;
for i:=1 to N do
  begin
  while not eoln do   {následníci vrcholu číslo i}
    begin
    read(E[j]);
    j:=j+1
    end;
  readln;
  V[i+1]:=j
  end;

{Inicializace:}
for i:=1 to N do
  Komponenta[i]:=0;   {žádný vrchol ještě není zařazen}
Barva:=0;
Zacatek:=1;
Navstiven:=0;

{Hledání komponent:}
while Navstiven <> N do  {existuje nenavštívený vrchol}
  begin
  Barva:=Barva+1;          {další komponenta souvislosti}
  while Komponenta[Zacatek]<>0 do Zacatek:=Zacatek+1;
  Vrchol:=1;
  Zasobnik[1]:=Zacatek;  {začátek další komponenty}
  Komponenta[Zacatek]:=Barva;
  Navstiven:=Navstiven+1;
  while Vrchol>0 do        {zásobník není prázdný}
    begin
    j:=Zasobnik[Vrchol];
    Vrchol:=Vrchol-1;     {odebrat vrchol zásobníku}
    for i:=V[j] to V[j+1]-1 do
     if Komponenta[E[i]]=0 then {následník E[i] nenavštíven}
        begin
```

```
        Komponenta[E[i]]:=Barva;
        Vrchol:=Vrchol+1;
        Zasobnik[Vrchol]:=E[i];
        Navstiven:=Navstiven+1
        end
    end
  end;

{Výpis výsledků:}
writeln('Komponenty souvislosti daného grafu jsou tvořeny ',
        'těmito vrcholy');
writeln('     (na každém řádku jedna komponenta):');
for j:=1 to Barva do
  begin
  for i:=1 to N do
    if Komponenta[i]=j then write(i:5);
  writeln
  end

end.
```

Všimněte si jedné zajímavé a poučné věci v právě uvedené programové ukázce. Vlastní jádro programu, kde dochází k určování komponent souvislosti, je tvořeno třemi do sebe vnořenými cykly. Každý z nich bude proveden nejvýše $N$-krát:

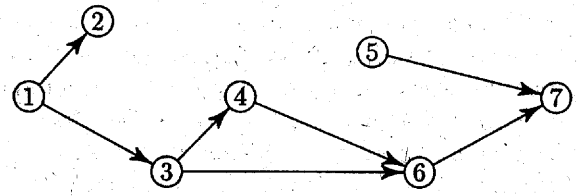
- vnější while-cyklus se opakuje pro každou komponentu a těch je v grafu o $N$ vrcholech nejvýše $N$
- vnitřní while-cyklus se může provádět až tolikrát, kolik vrcholů může být obsaženo v jedné komponentě, což je $N$
- for-cyklus je průchodem přes všechny následníky zvoleného vrcholu, každý vrchol může mít až $N$ následníků

Z tohoto pohledu na strukturu programu a z provedené úvahy o maximální délce jednotlivých cyklů můžeme usoudit, že program má časovou složitost $O(N^3)$. Přitom ale již dříve jsme v popisu algoritmu ukázali, že jeho časová složitost je kvadratická. Odhad $O(N^3)$ získaný pouze na základě pohledu na formální strukturu programu a bez hlubší analýzy algoritmu nebyl sice chybný, ale ukázal se zbytečně hrubý a nepřesný. Tři cykly vnořené v programu do sebe ještě neznamenají, že by jeho asymptotická časová složitost nemohla být kvadratická!



## 9.3 Topologické uspořádání

Acyklickým grafem nazýváme takový orientovaný graf, který neobsahuje žádný cyklus, tzn. neexistuje v něm cesta vycházející z nějakého vrcholu a končící v témže vrcholu. Vrcholy acyklického grafu lze vždy očíslovat tak, aby každá hrana vedla z vrcholu s nižším číslem do vrcholu s vyšším číslem. Toto uspořádání vrcholů se nazývá **topologické uspořádání** grafu a algoritmus, který ho vytváří, označujeme jako **topologické třídění**. Daný acyklický graf může mít více různých topologických uspořádání vrcholů. Pokud například graf neobsahuje žádnou hranu, potom každé uspořádání jeho vrcholů je topologické. Jestliže zkoumaný graf není acyklický, pak topologické uspořádání jeho vrcholů nemůže existovat.




*Obr. 24* Topologicky uspořádaný acyklický graf

V této kapitole si předvedeme algoritmus topologického třídění vrcholů daného orientovaného grafu. Součástí algoritmu bude ověření, zda je zpracovávaný graf skutečně acyklický. Pokud ne, algoritmus tuto skutečnost ohlásí. V opačném případě algoritmus určí jedno topologické uspořádání vrcholů grafu.

Problematika topologického třídění není úzce vázána jen na grafové úlohy. V různých podobách se objevuje při řešení nejrozmanitějších problémů. Její podstatou je rozšíření zadaného částečného uspořádání nějaké množiny na uspořádání úplné se zachováním všech původních relací. U grafů je touto množinou množina všech vrcholů daného grafu a zadané částečné uspořádání na ní je určováno souborem všech orientovaných hran grafu. Výsledné topologické uspořádání grafu je pak oním požadovaným zúplněním původního částečného uspořádání. Základní množinou může ale být třeba množina určitých činností, které je třeba vykonat, a dané částečné uspořádání určuje povinné návaznosti mezi činnostmi (kterou činnost lze začít provádět až po ukončení nějaké jiné). Úkolem

je určit takové pořadí všech činností, v jakém je možné postupovat, aby byly respektovány všechny povinné návaznosti.

Algoritmus topologického třídění je založen na skutečnosti, že v každém acyklickém grafu musí existovat nejméně jeden vrchol, do něhož nevede žádná hrana. Kdyby tomu tak nebylo, mohli bychom zvolit libovolný vrchol $v_1$, do něj vede hrana z jistého vrcholu $v_2$, do něj z vrcholu $v_3$ atd. Protože graf má jen konečně mnoho vrcholů, jednou se v této posloupnosti musí některý vrchol zopakovat. V posloupnosti vrcholů bychom tím ale získali cyklus, což je ve sporu s předpokladem, že graf je acyklický. Pokud tedy zjistíme, že ve zkoumaném grafu neexistuje žádný vrchol, do něhož nevede hrana, algoritmus topologického třídění může ihned ohlásit, že graf není acyklický, a nelze ho proto topologicky uspořádat.

Topologické třídění probíhá podle následujícího schématu. Najdeme v grafu jeden (libovolný) vrchol, do kterého nevede žádná hrana. Pokud by takový vrchol neexistoval, graf nelze topologicky uspořádat a algoritmus končí. Nalezený vrchol jistě může stát ve výsledném uspořádání na prvním místě. Zařadíme ho tedy na začátek vytvářeného topologického uspořádání vrcholů a z grafu ho vypustíme i se všemi hranami, které z něj vedou (všechny tyto hrany byly řádně respektovány a nadále již nejsou zajímavé). Pokud byl původní graf acyklický, je i zbytek grafu po vypuštění prvního zvoleného vrcholu acyklický. Uplatníme na něj proto stejný postup jako na původní graf, čímž získáme v pořadí druhý vrchol hledaného uspořádání. Stejně postupujeme dále až do úplného vyčerpání všech vrcholů grafu.

Pro programovou realizaci algoritmu bude výhodné znát u každého vrcholu grafu počet jeho předchůdců a seznam jeho následníků. Počet předchůdců využíváme pro nalezení vrcholu, do kterého nevede žádná hrana. Seznam následníků je zase zapotřebí při vypouštění vrcholu z grafu, kdy musíme všem následníkům vypouštěného vrcholu snížit o 1 uložený počet jejich předchůdců. Graf si proto budeme reprezentovat nejlépe pomocí seznamu následníků jednotlivých vrcholů, použít bychom mohli popř. i matici sousednosti. Dále použijeme pomocné pole $P$ indexované čísly vrcholů od 1 do $N$, do něhož budeme ukládat počet předchůdců každého vrcholu. Toto pole není v principu nezbytné, počet předchůdců jednotlivých vrcholů bychom mohli v každém okamžiku určit i ze zvolené vnitřní reprezentace grafu. Takové opakované výpočty by však zbytečně zpomalovaly celý algoritmus a vedly by ke zhoršení jeho časové složitosti.



Dalšího zrychlení dosáhneme zavedením seznamu $Q$, do kterého budeme průběžně ukládat čísla všech vrcholů s nulovým počtem předchůdců. Rovněž seznam $Q$ není nutný, vrcholy bez předchůdců můžeme velmi snadno vyhledávat v poli $P$. Vyhledání jednoho vrcholu v poli $P$ však vyžaduje provedení až $N$ operací, zatímco k vyzvednutí vrcholu ze seznamu $Q$ postačí čas konstantní.

Algoritmus topologického třídění má časovou složitost $O(N^2)$, kde $N$ je počet vrcholů grafu. Probíhá v $N$ krocích, přičemž v každém z nich je do výsledného uspořádání zařazen jeden vrchol. Každý krok výpočtu představuje nalezení vrcholu s nulovým počtem předchůdců (díky seznamu $Q$ konstantní čas) a vyřazení tohoto vrcholu spojené s provedením opravy počtu předchůdců všem jeho následníkům (těch je nejvýše $N$). V rámci těchto $N$ kroků se při opravách počtu předchůdců každá hrana grafu použije pouze jednou. Výslednou časovou složitost proto můžeme odhadnout přesněji výrazem $O(N + M)$, kde $M$ je počet hran grafu.

Celý postup topologického třídění si nyní zapíšeme ještě v Pascalu. Pro zjednodušení méně zajímavých částí programu budeme opět předpokládat, že na vstupu je graf zadán počtem vrcholů $N$ a pak přímo seznamy následníků jednotlivých vrcholů v pořadí od vrcholu číslo 1 k vrcholu $N$ (následníci jednoho vrcholu vždy na jednom řádku). Program pro jednoduchost nekontroluje korektnost vstupních dat (zda je počet vrcholů a hran grafu dostatečně malý s ohledem na definici konstant v programu, zda jsou čísla všech vrcholů z povoleného rozmezí). Naproti tomu součástí algoritmu je kontrola, zda je zadaný graf skutečně acyklický.

```
program TopologickeTrideni;

const MaxVrch = 100;        {max. přípustný počet vrcholů}
      MaxVrchPlus1 = 101;   {MaxVrch+1}
      MaxHran = 1000;       {max. přípustný počet hran}
      MaxHranPlus1 = 1001;  {MaxHran+1}

var V: array[1..MaxVrchPlus1] of 1..MaxHranPlus1;
    E: array[1..MaxHran] of 1..MaxVrch;
         {uložení grafu ve tvaru seznamu následníků}
    P: array[1..MaxVrch] of integer;
         {pro každý vrchol počet jeho předchůdců}
    Q: array[1..MaxVrch] of integer;
         {seznam čísel vrcholů nemajících předchůdce}
```

```
    PocetQ: 0..MaxVrch;
        {aktuální počet záznamů v seznamu Q}
    U: array[1..MaxVrch] of integer;
        {výsledné topologické uspořádání vrcholů grafu}
    N: 1..MaxVrch;          {počet všech vrcholů grafu}
    Cyklus: boolean;        {nalezen cyklus v grafu}
    i,j,k: integer;

begin
{Načtení vstupních dat - viz text výše:}
write('Počet vrcholů grafu: ');
readln(N);
writeln('Po řádcích následníci jednotlivých vrcholů',
        ' od 1 do ', N, ':');
V[1]:=1;
j:=1;
for i:=1 to N do
  begin
  while not eoln do                {následníci vrcholu číslo i}
    begin
    read(E[j]);
    j:=j+1
    end;
  readln;
  V[i+1]:=j
  end;

{Inicializace - počáteční obsazení polí P, Q:}
for i:=1 to N do
  P[i]:=0;                  {zatím vynulovat počet předchůdců}
for i:=1 to V[N+1]-1 do
  {prvek E[V[N+1]-1] představuje posledního následníka v E}
  begin
  j:=E[i];                  {číslo koncového vrcholu nějaké hrany}
  P[j]:=P[j]+1              {vrchol číslo j má dalšího předchůdce}
  end;
PocetQ := 0;                {ještě neznáme vrcholy bez předchůdců}
for i:=1 to N do
  if P[i]=0 then
    begin
    PocetQ := PocetQ + 1;
    Q[PocetQ] := i {další vrchol bez předchůdců}
```



```
    end;

{Vlastní topologické třídění:}
Cyklus := false;                {zatím nebyl nalezen cyklus}
i := 1;                                 {hledáme vrchol V[i]}
while not Cyklus and (i<=N) do
  if PocetQ = 0 then
    Cyklus := true                      {graf není acyklický}
  else
    begin
    U[i] := Q[PocetQ];          {další vrchol do uspořádání}
    PocetQ := PocetQ - 1;
    for j:=V[U[i]] to V[U[i]+1]-1 do
      begin
      {vrcholy E[j] jsou všichni následníci vrcholu U[i]}
      k := E[j];
      P[k] := P[k] - 1;       {snížíme jim počet předchůdců}
      if P[k] = 0 then      {nemá předchůdce - zařadit do Q}
        {vrcholy již vyřazené z grafu budou mít P[k] < 0}
        begin
        PocetQ := PocetQ + 1;
        Q[PocetQ] := k
        end
      end;
    i:=i+1
    end;

{Výpis výsledku:}
if Cyklus then
  begin
  writeln('Zadaný graf není acyklický');
  writeln('Nelze ho proto topologicky uspořádat')
  end
else
  begin
  writeln('Topologické uspořádání vrcholů grafu:');
  for i:=1 to N do write(U[i]:4);
  writeln
  end

end.
```

Problematika topologického třídění je podrobně zpracována také v knize [19], kde můžete nalézt jiný způsob programové realizace téhož algoritmu pomocí dynamických datových struktur.

## 9.4 Hledání nejkratší cesty

Nalezení nejkratší cesty v grafu je klasickou úlohou teorie grafů. Tuto úlohu je možné zformulovat různými způsoby. V nejjednodušší podobě je dán výchozí a cílový vrchol grafu s úkolem nalézt nejkratší cestu v grafu z výchozího do cílového vrcholu. Obvykle používáme algoritmy řešící o něco obecnější úlohu: pro daný výchozí vrchol nalézt nejkratší cesty do všech ostatních vrcholů grafu. Používané algoritmy většinou pracují stejným způsobem pro grafy neorientované i orientované.

Při hledání nejkratší cesty rozlišujeme mezi grafy neohodnocenými a ohodnocenými. V případě neohodnocených grafů znamená „nejkratší cesta" cestu s nejmenším možným počtem hran. Určíme ji nejlépe průchodem grafu do šířky (viz také kap. 8.1). U ohodnocených grafů rozumíme nejkratší cestou takovou cestu v grafu, na níž je součet ohodnocení hran minimální. Jako ohodnocení hran se používají nejčastěji kladná nebo alespoň nezáporná čísla. Připustíme-li záporná ohodnocení hran, nesmí v grafu existovat žádný záporný cyklus, tj. cesta se záporným součtem ohodnocení hran, která začíná a končí ve stejném vrcholu. V grafu se záporným cyklem nemá vůbec smysl hledat nejkratší cestu. Jestliže je totiž možné dojít z výchozího vrcholu ke kterémukoli vrcholu záporného cyklu, potom k jakékoli cestě v grafu snadno nalezneme cestu kratší. Postupujeme při tom tak, že dojdeme k zápornému cyklu a ten obejdeme tolikrát dokola, aby součet ohodnocení dostatečně poklesl.

Pro určování nejkratší cesty v ohodnoceném grafu se používají různé speciální algoritmy. Liší se podle toho, co víme o zkoumaném grafu a o ohodnocení jeho hran. Nejznámější je **Dijkstrův algoritmus**, který je použitelný pro všechny grafy s nezáporným ohodnocením hran. Jeho modifikací získáme algoritmus pro určení nejkratší cesty v libovolném acyklickém grafu, bez ohledu na ohodnocení hran. Všechny uvedené algoritmy si v následujících odstavcích předvedeme. Poznamenejme ještě, že Dijkstrův algoritmus je možné použít i pro hledání nejkratší cesty v neohodnoceném grafu místo výše zmíněného prohledávání grafu do



šířky. Stačí pohlížet na neohodnocený graf tak, jako by byl ohodnocený a všechny jeho hrany měly stejnou hodnotu 1.

Zbývá ukázat, jaké výsledky nám vlastně zmíněné algoritmy dávají. Ve své nejjednodušší podobě určují pouze délku nejkratší cesty, ale nikoli cestu samotnou. Pokud chceme určit také to, kudy nejkratší cesta vede, doplníme algoritmus o evidenci „předchůdců", stejně jako tomu bylo u prohledávání do šířky (kap. 8.2). Jednoduchým zpětným průchodem od cílového vrcholu k výchozímu pak získáme vlastní cestu. Algoritmy určují vždy jen jednu z cest minimální délky. Pokud v grafu existuje více různých cest minimální délky, bude vybrána ta z nich, kterou algoritmus nalezne jako první. Je ovšem možné ohodnotit hrany grafu například dvěma různými hodnotami, podle první z nich hledat nejkratší cestu a z více různých cest minimální délky pak vybírat výslednou na základě nějakého dalšího kritéria vázaného na druhé ohodnocení hran. K takovéto variantě Dijkstrova algoritmu s více kritérii se vrátíme v příkladu na konci této kapitoly.

Nejprve budeme řešit úlohu nalezení nejkratší cesty v neohodnoceném grafu (orientovaném nebo neorientovaném). Pro řešení použijeme průchod grafem do šířky. K procházení budeme potřebovat rychle získávat následníky jednotlivých vrcholů, takže se hodí reprezentovat graf seznamem následníků nebo maticí sousednosti. K realizaci průchodu budeme dále potřebovat pracovní pole pro uložení fronty a evidenci, které vrcholy již byly navštíveny. Pro evidenci navštívených vrcholů použijeme pole $D$ indexované přímo čísly vrcholů. Budeme do něj ukládat délky minimální cesty do jednotlivých vrcholů. Takováto evidence nám umožňuje testovat, který vrchol již byl navštíven, aniž se tím zvýší časová složitost algoritmu. Při výše uvedené volbě datových struktur bude mít algoritmus průchodu grafem časovou složitost $O(N^2)$, kde $N$ je počet vrcholů grafu. Bude totiž třeba navštívit postupně až $N$ vrcholů a pro každý z nich projít všechny jeho následníky, kterých může být až $N$. Stejně jako v případě hledání komponent souvislosti (kap. 9.2) můžeme zjemnit odhad časové složitosti algoritmu na $O(N+M)$, kde $M$ je počet hran grafu.

Procházení grafem začne v zadaném výchozím vrcholu a skončí ve chvíli, kdy bude navštíven cílový vrchol cesty. Musíme pamatovat na případ, že by cílový vrchol nebyl z výchozího vrcholu vůbec dostupný. V takové situaci výpočet skončí po navštívení všech dostupných vrcholů

a vzniklá situace bude ohlášena. Program může být napsán také tak, že se vždy projde celou dostupnou částí grafu. V tomto případě budou nalezeny nejkratší cesty ne do jednoho cílového vrcholu, ale do všech vrcholů grafu. Po skončení průchodu nalezneme v poli $D$ informaci o délce nejkratší cesty z výchozího vrcholu do všech těch vrcholů, do nichž jsme při procházení došli.

Jestliže budeme chtít znát nejen délku nejkratší cesty, ale i kudy tato nejkratší cesta vede, zavedeme ještě jedno pole $P$ indexované čísly vrcholů. Hodnota $P[x]$ bude vyjadřovat číslo vrcholu, který je předchůdcem vrcholu $x$ na nějaké nejkratší cestě ze zvoleného výchozího vrcholu. Správnou hodnotu $P[x]$ stanovíme vždy zároveň s určením nejkratší vzdálenosti $D[x]$ pro vrchol s číslem $x$. Celou nejkratší cestu z výchozího do cílového vrcholu pak získáme po ukončení průchodu grafem v samostatné druhé fázi výpočtu. Tato druhá fáze je velmi jednoduchá, má lineární časovou složitost a je tvořena přímým průchodem od cílového vrcholu zpět k výchozímu pomocí hodnot uložených v poli $P$.

Popsaný postup nalezení nejkratší cesty v neohodnoceném grafu nyní zapíšeme pořádně ve tvaru kompletního programu v Pascalu. Stejně jako v předchozích programech budeme pro jednoduchost předpokládat, že vstupní data jsou v pořádku (program neprovádí žádné kontroly) a že graf je na vstupu zadán nejprve počtem svých vrcholů a pak pro každý vrchol na jednom řádku čísly jeho následníků. Bezprostředně za celkovým počtem vrcholů grafu jsou zde navíc na vstupu uvedena čísla výchozího a cílového vrcholu, mezi nimiž máme nalézt nejkratší cestu.

```
program MinCesta1;
{Nalezení nejkratší cesty v neohodnoceném grafu}

const MaxVrch = 100;          {max. přípustný počet vrcholů}
      MaxVrchPlus1 = 101;     {MaxVrch+1}
      MaxHran = 1000;         {max. přípustný počet hran}
      MaxHranPlus1 = 1001;    {MaxHran+1}

var V: array[1..MaxVrchPlus1] of 1..MaxHranPlus1;
    E: array[1..MaxHran] of 1..MaxVrch;
         {uložení grafu ve tvaru seznamu následníků}
    Fronta: array[1..MaxVrch] of 1..MaxVrch;
         {fronta vrcholů pro realizaci průchodu grafem}
```



```
    ZacFr, KonFr: 1..MaxVrch;       {začátek a konec fronty}
    D: array[1..MaxVrch] of integer;   {vzdálenosti vrcholů}
    P: array[1..MaxVrch] of 1..MaxVrch; {předchůdci vrcholů}
    N: 1..MaxVrch;                  {počet všech vrcholů grafu}
    Start, Cil: 1..MaxVrch;         {výchozí a cílový vrchol}
    EI: 1..MaxVrch;                 {pomocné pro uložení E[I]}
    i,j: integer;

begin

{Načtení vstupních dat:}
write('Počet vrcholů grafu: ');
readln(N);
write('Číslo výchozího a cílového vrcholu: ');
readln(Start, Cil);
writeln('Po řádcích následníci jednotlivých vrcholů',
        ' od 1 do ', N, ':');
V[1]:=1;
j:=1;
for i:=1 to N do
  begin
  while not eoln do   {následníci vrcholu číslo i}
    begin
    read(E[j]);
    j:=j+1
    end;
  readln;
  V[i+1]:=j
  end;

{Inicializace:}
for i:=1 to N do
  D[i]:=-1;          {žádný vrchol ještě není navštíven}
D[Start]:=0;
Fronta[1]:=Start;  {ve frontě je jen výchozí vrchol}
ZacFr:=1;
KonFr:=1;

{Průchod grafem do šířky:}
while (D[Cil]=-1) and (ZacFrn<=KonFr) do
  begin
```

```
  j:=Fronta[ZacFr];  {jdeme z vrcholu j}
  ZacFr:=ZacFr+1;
  for i:=V[j] to V[j+1]-1 do
    begin
    EI := E[i];       {následníci vrcholu j jsou E[i]}
    if D[EI]=-1 then
      begin
      D[EI]:=D[j]+1;
      P[EI]:=j;
      KonFr:=KonFr+1;
      Fronta[KonFr]:=EI
      end
    end
  end;

{Vypsání nejkratší cesty:}
if D[Cil]=-1 then
  writeln('Cílový vrchol cesty není dostupný',
          ' ze zadaného výchozího vrcholu!')
else
  begin
  writeln('Délka nejkratší cesty z vrcholu ', Start,
          ' do vrcholu ', Cil,': ', D[Cil]);
  writeln('Nejkratší cesta v obráceném pořadí vrcholů');
  writeln('(od cílového vrcholu k výchozímu):');
  i:=Cil;
  while i<>Start do
    begin write(i:5); i:=P[i] end;
  writeln(Start:5)
  end

end.
```

Nejkratší cestu mezi dvěma vrcholy hledáme častěji v ohodnocených grafech. Typický je případ nezáporného ohodnocení hran a použití Dijkstrova algoritmu. Budeme tedy řešit například následující „silniční" úlohu: Mezi $N$ městy označenými čísly od 1 do $N$ je vybudována silniční síť. Některé dvojice měst jsou spojeny přímou silnicí, silnice se nekříží nikde mimo města. Jsou známy délky všech silnic v kilometrech. Najděte nejkratší silniční spojení z daného výchozího města do cílového města.



Pokud existuje více různých cest minimální možné délky, určete jednu libovolnou z nich.

Dijkstrův algoritmus postupně prochází dostupnými vrcholy grafu počínaje výchozím vrcholem. Způsob procházení je řízen průběžně počítanými hodnotami, které jsou přiřazovány jednotlivým vrcholům. Tyto hodnoty určují délku nejkratší cesty, jakou jsme dosud pro daný vrchol našli. Pro každý vrchol rozlišujeme, zda je jemu přiřazená hodnota dočasná (možná se ještě změní, tj. vylepší, zmenší), nebo zda je již trvalá (určuje výslednou délku nejkratší cesty do tohoto vrcholu). Před zahájením průchodu grafem bude mít výchozí vrchol přiřazenu dočasnou hodnotu 0 (cesta do něj má nulovou délku) a všechny ostatní vrcholy grafu mají dočasnou hodnotu „nekonečno" (zatím do nich neznáme žádné kratší cesty). Celý výpočet pak bude probíhat po krocích tak dlouho, dokud nebude cílovému vrcholu stanovena trvalá hodnota, popř. dokud nebude přiřazena trvalá hodnota všem dostupným vrcholům. V každém kroku výpočtu se provedou tyto akce:

1. Určíme vrchol s nejmenší dočasnou hodnotou (nechť je to vrchol s číslem $x$).

2. Dočasnou hodnotu tohoto vrcholu prohlásíme za trvalou.

3. Všem jeho následníkům, kteří mají ještě dočasnou hodnotu, přepočítáme jejich hodnoty podle předpisu

$$h_i = \min(h_i, h_x + d_{xi}),$$

kde $d_{xi}$ je délka silnice vedoucí z vrcholu $x$ do vrcholu $i$. Snížíme tedy všechny ty dočasné hodnoty vrcholů, které je možné snížit díky cestě vedoucí přes vrchol $x$.

Uvedený postup je jistě konečný, neboť v každém kroku je jednomu vrcholu přiřazena trvalá hodnota. Výpočet tedy skončí nejvýše po $N$ krocích, kde $N$ je počet vrcholů grafu. V každém kroku se musí vyhledat vrchol s nejmenší dočasnou hodnotou, což je výběr z $N$ vrcholů. Provede se při tom tudíž přibližně $N$ operací. Dále se přepočítávají dočasné hodnoty všech následníků vybraného vrcholu, kterých může být nejvýše $N$. Celkem se tedy vykoná nejvýše počet operací úměrný $N^2$. Jinými slovy řečeno, Dijkstrův algoritmus má kvadratickou časovou složitost.

Správnost algoritmu plyne z toho, jak je vybírán vrchol, jehož hodnota je prohlášena za trvalou. Je to vrchol s momentálně nejmenší dočasnou hodnotou. Tato dočasná hodnota představuje délku takové nejkratší

cesty do vrcholu, která vede pouze přes vrcholy s již trvalou hodnotou. Ta už rozhodně nepůjde vylepšit, neboť všechny ostatní vrcholy grafu mají dočasnou hodnotu větší (nebo stejnou) a graf má nezáporně ohodnocené hrany. Jakákoli jiná cesta do vybraného vrcholu by tedy byla delší, popř. stejně dlouhá. Je proto korektní prohlásit hodnotu vybraného vrcholu za trvalou.

Nyní se ještě zdržíme několika malými poznámkami k programové realizaci Dijkstrova algoritmu. Nejvýhodnější reprezentací ohodnoceného grafu pro potřeby tohoto algoritmu je asi seznam následníků jednotlivých vrcholů doplněný o ohodnocení hran (viz kap. 9.1, bod 4 a cvičení 1 na konci kap. 9). Práci se seznamem následníků jsme si ale již ukázali v předchozím programu, a proto nyní zvolíme jinou reprezentaci grafu. V následující programové ukázce budeme zadaný ohodnocený graf reprezentovat maticí vzdáleností. Pro uložení hodnot vrcholů použijeme pole přímo indexované čísly vrcholů. Evidenci dočasně a trvale ohodnocených vrcholů můžeme vést různě. Můžeme použít pole logických hodnot indexované čísly vrcholů, bylo by také možné místo toho využít pascalský datový typ množina a udržovat si množinu čísel dočasně ohodnocených vrcholů.

Po ukončení průchodu grafem udávají trvalé hodnoty vrcholů informaci o délce nejkratší cesty vedoucí do nich z výchozího vrcholu. Pokud budeme určovat i to, kudy cesta vede, použijeme stejnou úpravu jako v případě neohodnocených grafů. Opět zavedeme další pole $P$ indexované čísly vrcholů, do něhož budou ukládána čísla předchůdců jednotlivých vrcholů na nejkratší cestě. Hodnoty $P[x]$ budou určovány vždy zároveň se změnou dočasné hodnoty vrcholu $x$. Kdykoli je snížena dočasná hodnota vrcholu $x$ díky cestě přes vrchol y, položíme $P[x] = y$. Nejkratší cestu z výchozího do cílového vrcholu pak získáme na závěr výpočtu zpětným průchodem od cílového vrcholu k výchozímu pomocí hodnot uložených v poli $P$.

Následující program v Pascalu pro jednoduchost předpokládá, že vstupní data jsou správná, a nekontroluje je. Očekává vstupní data v tomto tvaru: počet vrcholů, číslo výchozího vrcholu, číslo cílového vrcholu, seznam ohodnocených hran grafu ve tvaru trojic $(i, j, d)$ s významem „existuje hrana z vrcholu $i$ do vrcholu $j$ délky $d$". Všechny hrany jsou chápány jako orientované.



```
program MinCesta2;
{Nalezení nejkratší cesty v ohodnoceném grafu}
{Dijkstrův algoritmus}

const MaxVrch = 100;      {max. přípustný počet vrcholů}

var Vzdal: array[1..MaxVrch,1..MaxVrch] of integer;
        {uložení grafu ve tvaru matice vzdáleností}
    H: array[1..MaxVrch] of integer;       {hodnoty vrcholů}
    Docas: array[1..MaxVrch] of boolean;   {dočasná hodnota}
    P: array[1..MaxVrch] of 1..MaxVrch; {předchůdci vrcholů}
    N: 1..MaxVrch;                {počet všech vrcholů grafu}
    Start, Cil: 1..MaxVrch;       {výchozí a cílový vrchol}
    Pruchod: boolean;             {pokračovat v průchodu}
    i,j: integer;

begin

{Načtení vstupních dat a inicializace:}
write('Počet vrcholů grafu: ');
readln(N);
write('Číslo výchozího a cílového vrcholu: ');
readln(Start, Cil);
for i:=1 to N do
  begin
  H[i]:=maxint;
  Docas[i]:=true;
  for j:=1 to N do Vzdal[i,j]:=maxint
  end;
writeln('Hrany grafu ve tvaru "odkud, kam, délka":');
while not eof do
  begin
  read(i,j);
  read(Vzdal[i,j])
  end;
H[Start]:=0;
Pruchod:=true;

{Průchod grafem:}
```

```
while Docas[Cil] and Pruchod do
  begin
  j:=Cil;      {hledáme vrchol s minimální dočasnou hodnotou}
  for i:=1 to N do
    if Docas[i] and (H[i]<H[j]) then j:=i;
  {vybrán vrchol j}
  if H[j]=maxint then
    Pruchod:=false    {Cil je nedostupný}
  else
    begin
    Docas[j]:=false;  {vrchol j má trvalou hodnotu}
    for i:=1 to N do
      if Vzdal[j,i] < maxint then
        if H[j]+Vzdal[j,i] < H[i] then
          begin
          H[i]:=H[j]+Vzdal[j,i]; {kratší cesta do vrcholu i}
          P[i]:=j                {novým předchůdcem i je j}
          end
    end
  end;

{Vypsání nejkratší cesty:}
if Docas[Cil] then
  writeln('Cílový vrchol cesty není dostupný',
          ' ze zadaného výchozího vrcholu!')
else
  begin
  writeln('Délka nejkratší cesty z vrcholu ', Start,
          ' do vrcholu ', Cil,': ', H[Cil]);
  writeln('Nejkratší cesta v obráceném pořadí vrcholů');
  writeln('(od cílového vrcholu k výchozímu):');
  i:=Cil;
  while i<>Start do
    begin write(i:5); i:=P[i] end;
  writeln(Start:5)
  end

end.
```

Drobnou úpravou Dijkstrova algoritmu získáme jiný postup pro nalezení nejkratší cesty, který lze použít i v případě záporně ohodnocených



hran. Je ovšem určen pouze pro acyklické grafy. Z předchozí kapitoly již víme, že vrcholy acyklického grafu lze vždy očíslovat tak, aby každá hrana vedla z vrcholu s nižším číslem do vrcholu s vyšším číslem. Pro tuto chvíli budeme předpokládat, že ve zkoumaném grafu máme již takto uspořádané a očíslované vrcholy. Pokud by tomu tak nebylo, museli bychom si je předem uspořádat metodou topologického třídění (viz kap. 9.3).

Výsledný algoritmus pro acyklické grafy se liší od původního Dijkstrova algoritmu pouze tím, jak vybírá v každém kroku výpočtu vrchol, jehož dočasná hodnota má být prohlášena za trvalou. Takový vrchol se nevybírá na základě minimality jeho dočasné hodnoty, ale místo toho se berou vrcholy grafu popořadě za sebou podle jejich očíslování (počínaje výchozím vrcholem a konče cílovým vrcholem cesty). Korektnost postupu vyplývá ze skutečnosti, že dočasná hodnota takto zvoleného vrcholu již nemůže být nikdy později snížena. Ke snížení jeho hodnoty by mohlo dojít pouze cestou vedoucí přes nějaký další, dosud neprozkoumaný vrchol. Vzhledem ke způsobu uspořádání vrcholů však již ze žádného dalšího vrcholu nevede cesta zpět do vrcholu právě vybraného. Algoritmus má stejnou kvadratickou časovou složitost jako algoritmus Dijkstrův, pokud ji vyjadřujeme pouze v závislosti na počtu vrcholů $N$. Přesnějším odhadem časové složitosti algoritmu je $O(N+M)$, kde $M$ je počet hran grafu. Z kap. 9.3 víme, že také algoritmus topologického uspořádání grafu má časovou složitost $O(N+M)$. V tomto čase tedy dokážeme vyřešit celou úlohu nalezení nejkratší cesty v acyklickém grafu, a to i v případě, že graf není předem topologicky uspořádán.

Programová realizace tohoto algoritmu je obdobná jako u Dijkstrova algoritmu, a proto se jí zde již nebudeme podrobněji zabývat.

Více pozornosti budeme raději věnovat jiné modifikaci Dijkstrova algoritmu, a to jeho doplnění o výběr nejlepší cesty podle více kritérií. Na následující úloze si ukážeme, o jaký problém půjde: Mezi $N$ městy označenými čísly od 1 do $N$ je vybudována síť autobusových linek. Každá linka spojuje některou dvojici měst, autobusy staví pouze na konečných stanicích. Mezi některými dvojicemi měst nemusí přímá autobusová linka existovat. O každé lince je známa doba jízdy autobusu a cena jízdného (kladná čísla). Pro danou dvojici měst (výchozí a cílové) určete nejlevnější autobusové spojení z jednoho do druhého. Pokud existuje více různých

tras s minimální cenou jízdného, vyberte z nich tu, na níž je celková doba jízdy autobusem nejmenší.

V právě uvedené úloze máme nalézt nejlepší autobusové spojení mezi dvěma městy na základě dvou různých hledisek, ceny jízdného a doby jízdy. Přitom musí být samozřejmě určeno, které hledisko je prvořadé (v naší úloze to je jízdné) a které se uplatní až v druhém pořadí (zde doba jízdy). Jestliže situaci zobecníme, řešíme vlastně tento problém: Je dán ohodnocený graf, v němž je každá hrana ohodnocena dvěma odlišnými hodnotami. Úkolem je nalézt nejlepší cestu mezi danými dvěma vrcholy grafu na základě obou ohodnocení. Ohodnocení se přitom uplatňují v pevně daném pořadí, nejprve jedno a teprve v případě shody prvního ohodnocení u více různých cest rozhoduje druhé ohodnocení.

Úlohu budeme řešit pomocí Dijkstrova algoritmu, v němž se pro výběr vrcholů použije prvořadé ohodnocení grafu (cena jízdného). Pro každý vrchol si však budeme navíc průběžně počítat a pamatovat i druhou hodnotu vázanou na druhé ohodnocení hran grafu. Obě hodnoty přiřazené jednotlivým vrcholům budou mít následující význam. První hodnota vrcholu, kterou je řízen i průběh výpočtu, udává nejkratší cenu jízdného, za jakou se do tohoto vrcholu zatím dokážeme dostat. Druhá hodnota určuje nejkratší dobu jízdy, za kterou se do vrcholu zatím umíme dostat při použití trasy s dosud minimální známou cenou jízdného. Ještě jednou zdůrazněme, že druhá hodnota vrcholu nepředstavuje vůbec nejkratší dobu jízdy do vrcholu, jakou jsme dosud našli, ale udává nejkratší dobu jízdy pouze mezi těmi cestami, na nichž je minimální cena.

Popíšeme nyní, jak probíhá výpočet. Označme $c_{xy}$ cenu jízdného na lince z města $x$ do města $y$, $t_{xy}$ značí dobu jízdy této linky. Dále označme $hc_x$ a $ht_x$ první a druhou hodnotu přiřazenou městu $x$. Konečně $p_x$ bude představovat číslo města, které předchází městu $x$ na nejlepší cestě z výchozího města do města $x$. Na začátku výpočtu označíme všechna města jako dočasně ohodnocená. Výchozí město bude mít obě hodnoty $hc$ a $ht$ rovny nule (dostaneme se do něj bez placení a za nulový čas), všechna ostatní města budou mít počáteční hodnoty $hc$ a $ht$ nastaveny na „nekonečno“. Vlastní výpočet pak probíhá po krocích tak dlouho, dokud cílové město nezíská trvalé ohodnocení. V každém kroku výpočtu se provedou tyto akce:

1. určíme dočasně ohodnocené město s nejmenší hodnotou $hc$ (nechť je to město s číslem $x$)



2. město $x$ prohlásíme za trvale ohodnocené
3. všem jeho následníkům, kteří mají ještě dočasnou hodnotu, přepočítáme jejich hodnoty takto (nechť $i$ je index obecně jednoho z následníků města $x$):
   - jestliže $hc_x + c_{xi} < hc_i$ (našli jsme levnější cestu), položíme

     $$hc_i = hc_x + c_{xi} \text{ (nové menší jízdné do } i\text{)}$$
     $$ht_i = ht_x + t_{xi} \text{ (doba jízdy na nové trase)}$$
     $$p_i = x \text{ (nový předchůdce města } i\text{)}$$

   - jestliže $hc_x + c_{xi} = hc_i$ (našli jsme jinou stejně levnou cestu do města $i$, vybíráme proto podle druhého kritéria) a přitom $ht_x + t_{xi} < ht_i$ (nová cesta je rychlejší), položíme

     $$ht_i = ht_x + t_{xi} \text{ (doba jízdy na nové trase)}$$
     $$p_i = x \text{ (nový předchůdce města } i\text{)}$$

Zdůvodnění správnosti algoritmu i časová složitost jsou obdobné jako v základní verzi Dijkstrova algoritmu. Rovněž programová realizace je analogická. Uvádíme zde úplný program v Pascalu řešící naši úlohu s autobusovými linkami. Všimněte si, jak málo se liší od předcházejícího programu. Program opět pro jednoduchost nekontroluje správnost vstupních dat a předpokládá, že na vstupu je zadán počet měst, číslo výchozího a cílového města a dále seznam autobusových linek ve tvaru čtveřic $(i, j, c, t)$ s významem „autobusová linka mezi městy $i$, $j$ má jízdné $c$ a dobu jízdy $t$“. Všechny linky budeme chápat jako obousměrné (autobus jezdí tam i zpět stejně rychle a za stejnou cenu).

```
program MinCesta3;
{Nalezení nejkratší cesty v ohodnoceném grafu}
{Dijkstrův algoritmus s více kritérii}

const MaxVrch = 100;        {max. přípustný počet vrcholů}

var Cena, Cas: array[1..MaxVrch,1..MaxVrch] of integer;
          {ohodnocení hran grafu ve tvaru matic vzdáleností}
    HC, HT: array[1..MaxVrch] of integer;  {hodnoty vrcholů}
```

```
    Docas: array[1..MaxVrch] of boolean;   {dočasná hodnota}
    P: array[1..MaxVrch] of 1..MaxVrch; {předchůdci vrcholů}
    N: 1..MaxVrch;                {počet všech vrcholů grafu}
    Start, Cil: 1..MaxVrch;       {výchozí a cílové město}
    Pruchod: boolean;             {pokračovat v průchodu}
    NovaCena: integer;            {možné zlepšení HC vrcholu}
    NovyCas: integer;             {možné zlepšení HT vrcholu}
    i,j: integer;

begin

{Načtení vstupních dat a inicializace:}
write('Počet měst: ');
readln(N);
write('Číslo výchozího a cílového města: ');
readln(Start, Cil);
for i:=1 to N do
  begin
  HC[i]:=maxint;
  HT[i]:=maxint;
  Docas[i]:=true;
  for j:=1 to N do
    begin Cena[i,j]:=maxint; Cas[i,j]:=maxint end;
  end;
writeln('Autobusové linky ve tvaru ',
        '"odkud  kam  jízdné  čas":');
while not eof do
  begin
  read(i,j);
  read(Cena[i,j],Cas[i,j]);
  Cena[j,i]:=Cena[i,j];
  Cas[j,i]:=Cas[i,j]
  end;
H1[Start]:=0;
H2[Start]:=0;
Pruchod:=true;

{Průchod grafem:}
while Docas[Cil] and Pruchod do
  begin
  j:=Cil;      {hledáme vrchol s minimální dočasnou hodnotou}
```



```
  for i:=1 to N do
    if Docas[i] and (HC[i]<HC[j]) then j:=i;
  {vybrán vrchol j}
  if HC[j]=maxint then
    Pruchod:=false    {Cil je nedostupný}
  else
    begin
    Docas[j]:=false;  {vrchol j má trvalou hodnotu}
    for i:=1 to N do
      if Cena[j,i] < maxint then
        begin
        NovaCena := HC[j]+Cena[j,i];
        NovyCas := HT[j]+Cas[j,i];
        if NovaCena < HC[i] then
          begin
          HC[i]:=NovaCena;        {kratší cesta do vrcholu i}
          HT[i]:=NovyCas;         {čas na nové cestě do i}
          P[i]:=j                 {novým předchůdcem i je j}
          end
        else
          if NovaCena = HC[i] then
                {nalezena jiná cesta stejné minimální ceny}
            if NovyCas < HT[i] then  {kratší čas}
              begin
              HT[i]:=NovyCas;     {čas na nové cestě do i}
              P[i]:=j             {novým předchůdcem i je j}
              end
        end
    end
  end;

{Vypsání nejkratší cesty:}
if Docas[Cil] then
  writeln('Cílové město není autobusovými linkami dostupné',
          ' ze zadaného výchozího města!')
else
  begin
  writeln('Cena jízdného na nejlevnější cestě z města ',
          Start,' do města ', Cil,': ', HC[Cil]);
  writeln('Nejmenší možná doba jízdy na této nejlevnější ',
          'cestě: ', HT[Cil]);
```

```
    writeln('Vybraná nejlepší cesta v obráceném pořadí měst');
    writeln('(od cílového města k výchozímu):');
    i:=Cil;
    while i<>Start do
      begin write(i:5); i:=P[i] end;
    writeln(Start:5)
    end

end.
```

Podobným způsobem odvodíme další modifikace Dijkstrova algoritmu v případě jiných kritérií výběru nejlepší cesty. Můžeme použít i více než dvě kritéria. Některé z hledisek výběru nemusí být ani vázáno na zvláštní ohodnocení hran grafu, preference výběru cesty může být vyjádřena i jinak (například dát přednost cestě, která vede přes jistý vrchol, cestě s nejmenším počtem hran apod.). Příklady takových úloh naleznete ve cvičeních.

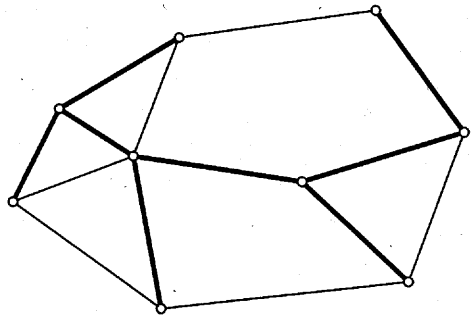
### 9.5 Minimální kostra grafu

Hledání minimální kostry grafu je dalším z klasických problémů teorie grafů. Úlohu si nejprve přiblížíme na příkladu s městy a se silnicemi. Mezi $N$ městy označenými čísly od 1 do $N$ je vybudována silniční síť. Každá silnice spojuje vždy dvojici měst a je známa její délka v kilometrech. Mezi některými dvojicemi měst přímá silnice nevede, ale z každého města je možné dojet po silnicích do libovolného jiného města (třeba i více různými způsoby). Všechny silnice jsou obousměrné a nekříží se nikde mimo města. Při velké sněhové bouři byly všechny silnice zaváty sněhem. Určete minimální celkovou délku silnic, z nichž je třeba odklidit sníh, aby byla všechna města v zemi navzájem pospojována sjízdnými silnicemi.

Tutéž úlohu zformulujeme nyní ještě jednou obecně v řeči teorie grafů. Silniční síť představuje souvislý neorientovaný ohodnocený graf, v němž vrcholy grafu odpovídají městům, hrany grafu silnicím a délky jednotlivých silnic jsou ohodnocením hran. Právě pro souvislé neorientované ohodnocené grafy řešíme úlohu nalézt minimální kostru grafu. **Kostrou** souvislého grafu rozumíme takovou množinu jeho hran, která obsahuje nejmenší možný počet hran, přičemž těmito hranami jsou po-



spojovány všechny vrcholy grafu tak, že mezi libovolnými dvěma z nich vede cesta po hranách z kostry. Kostra grafu nemůže obsahovat žádný cyklus. Jinak by totiž bylo možné vynechat z ní jednu libovolnou hranu, která je součástí cyklu, a všechny vrcholy grafu by pak zůstaly navzájem pospojovány. To by ovšem bylo ve sporu s podmínkou, že kostra obsahuje minimální možný počet hran. Mezi každou dvojicí vrcholů tedy vede právě jedna cesta po hranách kostry. Souvislý graf o $N$ vrcholech může mít více různých koster, každá z nich však obsahuje přesně $(N - 1)$ hran. V případě ohodnoceného grafu má smysl hovořit o minimální kostře grafu. **Minimální kostrou grafu** nazýváme kostru, u níž je součet ohodnocení všech hran minimální. Jeden graf může mít i více různých minimálních koster. V takovém případě budeme hledat jednu libovolnou z nich.



*Obr. 25* Kostra grafu

Různé algoritmy řešící tuto úlohu lze nalézt v každé základní učebnici diskrétní matematiky nebo teorie grafů. Můžeme použít například tzv. hladový algoritmus. Před jeho uvedením si ještě musíme uvědomit, že libovolnou kostru souvislého grafu získáme vynecháním co možná největšího počtu hran tak, aby graf zůstal souvislý. Při hledání minimální kostry grafu budeme vynechávat ty hrany, které mají vyšší ohodnocení. Postupujeme proto následovně. Všechny hrany daného grafu uspořádáme podle jejich ohodnocení vzestupně. Na pořadí hran stejné délky nezáleží. Potom postupně bereme jednotlivé hrany od nejkratší k nejdelší a pro každou z nich zkoumáme, zda ji můžeme zařadit do vytvářené kostry, tj. zda by jejím přidáním do vytvářené kostry nevznikl v kostře cyklus.

Pokud cyklus nevznikne, hranu do kostry zařadíme, v opačném případě ji vynecháme. Celý výpočet ukončíme ve chvíli, kdy jsme do kostry již zařadili potřebných $(N-1)$ hran.

K efektivnímu naprogramování uvedeného algoritmu je třeba zvolit vhodnou datovou reprezentaci. Graf je zadán seznamem hran a jejich ohodnocením. Vzhledem k tomu, že potřebujeme všechny hrany setřídit a pracovat potom s takto setříděným seznamem hran, použijeme tuto podobu i pro vnitřní reprezentaci grafu v programu. Ve stejném tvaru, tj. jako seznam hran, budeme vytvářet a ukládat (nebo přímo tisknout) i výslednou kostru. Zbývá nám poslední a nejtěžší úkol: Potřebujeme mít možnost během výpočtu snadným způsobem testovat, kterou hranu lze přidat do vytvářené kostry. Bylo by dost pracné a pomalé procházet vždy celý seznam hran již zařazených do kostry a nějak zkoumat, zda společně s právě zpracovávanou hranou netvoří cyklus. Lepší je použít vhodnou pomocnou datovou strukturu, v níž bude zaznamenáno, které vrcholy grafu jsou již pospojovány hranami zařazenými do kostry. Postupně vytvářená kostra je nesouvislým podgrafem původního grafu — obsahuje stejnou množinu vrcholů, ale méně hran. Na začátku výpočtu není v kostře ještě žádná hrana a každý vrchol grafu je izolovaný (tvoří vlastní komponentu souvislosti). Každým zařazením další hrany do kostry se o 1 sníží počet komponent souvislosti. Nově přidávaná hrana totiž musí spojovat dvojici vrcholů z dosud různých komponent, jinak by její zařazení do kostry vytvořilo cyklus. Po zařazení všech $(N-1)$ hran do kostry bude pospojováno všech $N$ vrcholů do souvislého grafu bez cyklů.

Evidenci průběžně vytvářených komponent souvislosti (tj. skupin již pospojovaných vrcholů) je možné programově realizovat různými způsoby. Jednoduchým a názorným řešením (i když z hlediska efektivity ne zcela nejlepším) je použít pomocné jednorozměrné pole velikosti $N$ indexované čísly vrcholů od 1 do $N$. V tomto poli bude pro každý vrchol uloženo číslo komponenty, do níž vrchol momentálně náleží. Při zpracování každé další hrany pak snadno otestujeme, zda její krajní vrcholy jsou již v téže komponentě souvislosti. Pokud ano, zkoumanou hranu vynecháme, v opačném případě ji zařadíme do kostry. Při zařazení nové hrany do kostry musíme samozřejmě opravit také naši evidenci pospojovaných vrcholů, neboť tím zároveň dojde k propojení dvou dosud samostatných komponent souvislosti. Všem vrcholům obou těchto komponent musíme



proto přiřadit stejné číslo komponenty souvislosti (například jedno z čísel právě spojovaných komponent).

Popsaný způsob práce s postupně vytvářenými komponentami souvislosti grafu odpovídá obecné datové struktuře nazývané **faktorová množina**. Faktorovou množinou rozumíme takovou $N$-prvkovou množinu, která je rozdělena do několika disjunktních tříd (tzv. faktorových tříd), přičemž každý z $N$ prvků náleží do právě jedné z tříd. S faktorovou množinou provádíme dvě základní operace: určení, do které faktorové třídy patří daný prvek, a sjednocení dvou tříd. Každé sjednocení sníží počet faktorových tříd o 1. Nejjednodušší implementaci faktorové množiny jsme popsali v předchozím odstavci. Spočívá v použití pole, které je přímo indexováno prvky množiny. Hodnota přiřazená v poli každému prvku určuje třídu, ve které tento prvek leží. Nalezení faktorové třídy daného prvku je pak triviální a má konstantní časovou složitost. Operace sjednocení dvou tříd vyžaduje vyhledat v poli všechny prvky náležející do jedné z těchto tříd a změnit jim hodnotu uloženou v poli. Složitost této operace je tedy $O(N)$. Existuje i efektivnější způsob implementace faktorové množiny, který je založen na reprezentaci každé faktorové třídy pomocí stromu. Tuto metodu můžete nalézt v odborné literatuře (např. v [13]).

Výše uvedené řešení problému nalezení minimální kostry grafu si nyní ukážeme naprogramované v Pascalu. Vstupem programu je počet měst $N$ a dále seznam všech silnic vedoucích mezi městy. Každá silnice je určena trojicí čísel: čísly obou měst silnicí spojených a délkou silnice. Ke zde uvedenému ukázkovému programu je třeba ještě poznamenat několik maličkostí. Úvodní cyklus „`while not eof do`“ pro načítání vstupních dat proběhne korektně pouze v některých implementacích jazyka Pascal. Je k tomu totiž třeba zajistit, aby bezprostředně po zadání posledního vstupního údaje následoval na vstupu příznak konce vstupního souboru. V Turbo Pascalu splníte tento požadavek snadno. Není samozřejmě žádným problémem v případě potřeby program mírně upravit a ukončit cyklus pro načítání vstupních dat jiným způsobem, např. zadáním nějaké zvláštní hodnoty přímo na vstupu. Program pro jednoduchost nekontroluje správnost vstupních dat a předpokládá, že všechny vstupní údaje jsou zadány bezchybně. Po přečtení vstupních dat program musí setřídit seznam hran grafu podle velikosti. V našem programu jsme pro zjednodušení a zkrácení zápisu použili ten nejjednodušší třídicí algoritmus,

a to tzv. třídění přímým výběrem. Lepší a rychlejší třídicí algoritmy jsou popsány v kap. 11.1.

```
program MinimalniKostra;
{Určení minimální kostry souvislého
 ohodnoceného neorientovaného grafu}

const MaxVrch = 100;     {maximální počet vrcholů grafu}
      MaxHran = 1000;    {maximální počet hran grafu}
type Hrana = record
             V1, V2: integer;   {spojené vrcholy}
             Delka: integer     {ohodnocení hrany}
             end;

var Graf: array[1..MaxHran] of Hrana;          {uložení grafu}
    Komponenta: array[1..MaxVrch] of integer;  {komponenty}
    N: 1..MaxVrch;              {skutečný počet vrcholů grafu}
    PomHrana: Hrana;
    K1,K2: integer;
    S,I,J,K: integer;

begin

{Načtení vstupních dat:}
write('Počet vrcholů grafu: ');
readln(N);
writeln('Hrany grafu:  ODKUD  KAM  DELKA');
S:=0;
while not eof do
  begin
  S:=S+1;           {počet všech hran grafu}
  with Graf[S] do read(V1, V2, Delka)
  end;

{Setřídění hran podle velikosti od nejkratší k nejdelší
 - pro jednoduchost zde použijeme jednoduchý třídicí
 algoritmus třídění přímým výběrem:}
for I:=1 to S-1 do
  begin
  K:=I;
  for J:=I+1 to S do
```



```
    if Graf[J].Delka<Graf[K].Delka then K:=J;
  if K<>I then
    begin
    PomHrana:=Graf[K];
    Graf[K]:=Graf[I];
    Graf[I]:=PomHrana
    end
  end;

{Vlastní určení minimální kostry grafu
 - nalezená kostra se neukládá, přímo se tiskne:}
writeln;
writeln('Hrany tvořící minimální kostru grafu:');
for I:=1 to N do Komponenta[I]:=I;
  {počáteční jednoprvkové komponenty souvislosti}
I:=0;
K:=0;                {počet hran zařazených do kostry}
while K<N-1 do
  begin
  I:=I+1;            {zkoumaná hrana}
  K1:=Komponenta[Graf[I].V1];
  K2:=Komponenta[Graf[I].V2];
  if K1<>K2 then     {hranu I zařadíme do kostry}
    begin
    K:=K+1;
    writeln(Graf[I].V1:5,Graf[I].V2:5);
    for J:=1 to N do
      if Komponenta[J]=K2 then Komponenta[J]:=K1
    end
  end

end.
```

Důkaz správnosti uvedeného algoritmu je matematicky trochu obtížnější. Postup výpočtu je jistě konečný, neboť nejprve se setřídí podle velikosti konečně mnoho hran a potom se vytváří kostra postupným zkoumáním jednotlivých hran. Počet kroků tohoto procesu je tedy omezen počtem hran. Složitost testování každé hrany, zda ji můžeme přidat do kostry, je konstantní díky pomocné evidenci komponent souvislosti, přidání hrany do kostry pak vyžaduje projít pomocným polem, tj. provést

výpočet délky rovnající se počtu vrcholů. Odtud také přímo plyne časová složitost algoritmu. Označíme-li počet hran v grafu (tj. počet silnic) symbolem $M$, můžeme setřídění $M$ hodnot podle velikosti provést jednoduchým třídicím algoritmem složitosti $O(M^2)$, jako jsme to udělali v našem programu. Můžeme také použít lepší třídicí algoritmus, přinejlepším však s časovou složitostí $O(M \log M)$. (O složitosti třídicích algoritmů se více dočtete v kap. 11.) Vlastní hladový algoritmus potom vyžaduje provést maximálně $M$ kroků, z nichž každý má složitost $N$ (kde $N$ je počet vrcholů grafu).

Z uvedeného postupu řešení je zřejmé, že algoritmus vyhledá nějakou kostru zadaného grafu. Postupně totiž do vytvářené kostry zařazuje jen takové hrany, jejichž přidáním nevznikne cyklus, a zařadí jich tam správný počet $(N - 1)$. Zbývá ukázat, že nalezená kostra je skutečně minimální. Hrany tvořící nalezenou kostru označíme pro potřeby důkazu $e_1, e_2, \ldots, e_k$, kde $k = N - 1$, a to v pořadí, v jakém byly vybírány, tzn. podle velikosti od nejmenší k největší. Každý graf má jistě nějakou minimální kostru (může jich mít i více různých), neboť má konečně mnoho koster a z nich lze vybrat kostru s minimálním ohodnocením. Uvažujme jednu libovolnou minimální kostru, uspořádejme její hrany podle velikosti od nejmenší k největší a označme je po řadě $f_1, f_2, \ldots, f_k$. Dokážeme, že pro každé $i$ od 1 do $k$ platí, že ohodnocení hrany $e_i$ musí být menší nebo rovno ohodnocení hrany $f_i$. Odtud již přímo plyne, že součet ohodnocení všech hran $e_1, e_2, \ldots, e_k$ není větší než součet ohodnocení hran $f_1, f_2, \ldots, f_k$, a že tedy kostra nalezená naším algoritmem je minimální.

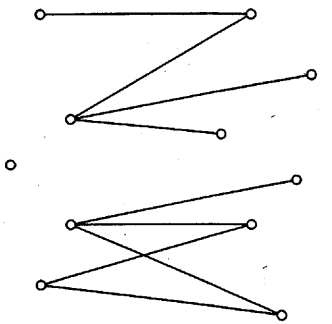
Důkaz provedeme sporem. Předpokládejme, že pro nějaké $i$ mezi 1 a $k$ platí, že ohodnocení $e_i$ je ostře větší než ohodnocení hrany $f_i$. Jistě $i > 1$, neboť za $e_1$ jsme v algoritmu vzali hranu s minimálním ohodnocením. Uvažujme množiny hran $E_{i-1} = \{e_1, \ldots, e_{i-1}\}$ a $F_i = \{f_1, \ldots, f_i\}$. Obě tyto množiny hran jsou podmnožinami nějakých koster, a proto neobsahují žádný cyklus. K dokončení důkazu stačí ukázat, že v množině $F_i$ existuje taková hrana $f$, že $E_{i-1} \cup \{f\}$ neobsahuje cyklus. Všechny hrany obsažené v množině $F_i$ mají totiž ostře menší ohodnocení než hrana $e_i$ (to plyne z předpokladu našeho důkazu a z uspořádání množiny $F_i$), tedy i hrana $f$ má ohodnocení menší než hrana $e_i$. Popsaný algoritmus nalezení minimální kostry by tudíž jako v pořadí $i$-tou zařadil do vytvářené kostry hranu $f$ místo hrany $e_i$, což je spor. Existence hrany $f \in F_i$ takové, že $E_{i-1} \cup \{f\}$ neobsahuje cyklus, přímo vyplývá ze



skutečnosti, že množina $F_i$ obsahuje o jednu hranu více než množina $E_{i-1}$. Uvážíme-li souvislé komponenty grafu určené množinou hran $E_{i-1}$ (žádná z těchto komponent neobsahuje cyklus), jistě existuje hrana z $F_i$ spojující dvě různé komponenty — a to je hledaná hrana $f$. V opačném případě by totiž každá hrana z $F_i$ musela spojovat dvojici vrcholů téže komponenty, a protože v $F_i$ je o jednu hranu více než v $E_{i-1}$, nutně by tak některá skupina hran z $F_i$ vytvořila cyklus.

## 9.6 Bipartitní grafy

Neorientovaný graf se nazývá **bipartitní**, jestliže je možné rozdělit všechny jeho vrcholy do dvou disjunktních skupin tak, aby každá hrana grafu spojovala vždy dvojici vrcholů náležejících do různých skupin. Řečeno jinými slovy, uvnitř ani jedné z obou skupin nesmí vést žádná hrana. Naším úkolem bude zjistit o daném grafu, zda je bipartitní. Pokud ano, chceme navíc nalézt odpovídající rozdělení vrcholů grafu do dvou skupin. Může existovat více různých takových rozdělení vrcholů, nás bude zajímat nalezení jednoho libovolného z nich.



*Obr. 26* Bipartitní graf

Úlohu o bipartitním grafu si můžeme opět převést do řeči měst a silnic. Mezi $N$ městy označenými čísly od 1 do $N$ je vybudována silniční síť. Každá silnice spojuje vždy dvojici měst, mezi některými dvojicemi měst přímá silnice nevede. Všechny silnice jsou obousměrné a nekříží se nikde mimo města. Určete, zda je možné rozdělit města do dvou skupin

tak, aby každá silnice spojovala vždy dvě města patřící do různých skupin. Nezáleží přitom na velikosti obou skupin, ale každé město musí být do jedné z nich zařazeno. Je-li takové rozdělení měst do skupin možné, najděte ho. Z více možných řešení vypište jedno libovolné.

Algoritmus řešení této úlohy trochu připomíná algoritmus na hledání komponent souvislosti grafu, který jsme uvedli v kap. 9.2. Je opět založen na vhodném procházení grafu. Stejně jako při určování komponent souvislosti můžeme použít libovolný způsob procházení, do hloubky nebo do šířky. Postup výpočtu si můžeme názorně představit tak, že při rozdělování měst do skupin budeme jednotlivá města „obarvovat" dvěma barvami podle toho, do které skupiny bylo město zařazeno. Městům zařazeným do první skupiny přiřadíme barvu 1 a městům z druhé skupiny barvu $-1$. Na začátku výpočtu není žádné město nijak obarveno (má barvu 0), neboť dosud nebylo zařazeno do žádné ze skupin.

Celý proces rozdělování měst do skupin zahájíme tím, že jedno libovolné město (například město s číslem 1) obarvíme barvou 1. Města, do kterých z tohoto města vede přímá silnice, nesmějí podle zadání patřit do stejné skupiny. Obarvíme je proto barvou $-1$. Podobně postupujeme stále dál, na této základní myšlence je založen celý algoritmus. Obecně platí, že bylo-li nějaké město zařazeno do jedné ze skupin, musí být všechna města, která s ním jsou spojena přímou silnicí, zařazena do opačné skupiny. Přitom musíme průběžně kontrolovat, zda nedojde ke konfliktu. Konflikt nastane tehdy, potřebujeme-li nějaké již obarvené město obarvit opačnou barvou. V takovém případě rozdělení měst do skupin neexistuje a výpočet ihned ukončíme. Jinak pokračujeme v obarvování měst tak dlouho, dokud jsme nuceni obarvovat další města (vlastně jako důsledek počátečního obarvení prvního města). Jestliže se tento proces obarvování zastaví a přitom ještě existují neobarvená města, znamená to, že graf není souvislý. Máme zatím obarvena města z jedné komponenty souvislosti. S obarvováním měst z další komponenty začneme opět od začátku. Zvolíme tedy jedno libovolné dosud neobarvené město (například to s nejmenším číslem) a obarvíme ho barvou 1. Dále pokračujeme v obarvování stejným způsobem. Celé rozdělování měst skončí ve chvíli, kdy jsou všechna města obarvena a zkontrolována, zda jejich obarvení nezpůsobuje žádný konflikt (obarvení měst pak určuje jedno možné rozdělení měst do skupin) nebo dojde-li při obarvování ke konfliktu (rozdělení měst v takovém případě neexistuje).



Pro efektivní naprogramování uvedeného postupu je důležitá vhodná volba datových struktur. K reprezentaci grafu můžeme použít nejlépe matici sousednosti nebo seznam následníků jednotlivých vrcholů. Zvolme tedy pro ukázku tentokrát na rozdíl od kap. 9.2 matici sousednosti. Informace o existujících silnicích uložíme do dvojrozměrného čtvercového pole logických hodnot velikosti $N \times N$. Pro evidenci obarvení měst, tj. zařazení měst do skupin, použijeme pomocné pole přímo indexované čísly měst od 1 do $N$. Neobarvená města budou mít v tomto poli přiřazenu hodnotu 0, obarvená pak hodnotu 1 nebo $-1$ podle zvolené barvy. Dále si potřebujeme udržovat seznam těch měst, která jsme již obarvili, ale dosud jsme nezajistili, aby města, do kterých z nich vede přímá silnice, patřila do opačné skupiny. Pro uložení tohoto seznamu použijeme jednorozměrné pole o $N$ prvcích. Na pořadí, v jakém budeme z tohoto seznamu vybírat prvky pro další zpracování, vůbec nezáleží. Typicky bude seznam obsluhován jako zásobník (pak půjde o průchod grafem do hloubky) nebo jako fronta (průchod grafem do šířky). Zásobník se snáze programuje, takže se přikloníme spíše k němu.

Z uvedeného postupu již plyne správnost algoritmu. Postupem obarvování je zajištěno, že se nikdy nemohou dostat do téže skupiny dvě města, mezi nimiž vede přímá silnice. Pokud tedy nalezneme bez konfliktu obarvení všech měst, určuje toto obarvení správné rozdělení měst do skupin. Naopak, konflikt nastane jedině ve chvíli, kdy by již dříve obarvené město mělo být nově zařazeno do opačné skupiny, než kam patří, tj. když rozdělení měst do skupin neexistuje. Konečnost výpočtu je dána tím, že v každém kroku je obarveno jedno město. Počet kroků výpočtu je tedy roven nejvýše počtu všech měst $N$. Odtud plyne, že časová složitost algoritmu je úměrná $N^2$, neboť každý krok výpočtu představuje provedení až $kN$ operací pro vhodnou konstantu $k$ (jeden průchod přes všechny následníky právě obarveného města).

Následující program v Pascalu předpokládá, že na vstupu je zadán počet měst a dále seznam všech silnic ve tvaru dvojic čísel (mezi kterými dvěma městy silnice vede). Pro zvýšení přehlednosti programu a zkrácení zápisu jsme se úmyslně dopustili stejných drobných nekorektností jako v dřívějších ukázkách: program nekontroluje správnost vstupních dat a předpokládá, že hned za posledním údajem na vstupu nalezne příznak konce vstupního souboru.

```
program BipartitniGraf;
{Rozdělení měst do dvou skupin - bipartitní graf}

const MaxN=100;                    {maximální možný počet měst}

var A:array[1..MaxN,1..MaxN] of boolean;     {matice silnic}
    Barva:array[1..MaxN] of -1..1;           {rozdělení měst}
    Zasob:array[1..MaxN] of 1..MaxN;  {pracovní seznam měst}
    Vrchol:0..MaxN;          {index posledního prvku seznamu}
    N:1..MaxN;               {skutečný počet měst}
    Zarazena:integer;        {počet měst zařazených do skupin}
    Konflikt:boolean;        {města nelze rozdělit}
    i,j:integer;

begin

{Inicializace proměnných a načtení vstupních dat:}
write('Počet měst: ');
readln(N);
for i:=1 to N do
  begin
  for j:=1 to N do A[i,j]:=false;
  Barva[i]:=0
  end;
write('Seznam všech silnic');
writeln(' - dvojice čísel měst ODKUD a KAM vede:');
while not eof do
  begin
  read(i,j);
  A[i,j]:=true;
  A[j,i]:=true    {silnice jsou obousměrné}
  end;

{Rozdělování měst:}
Zasob[1]:=1;     {výchozí město}
Vrchol:=1;       {v seznamu zařazených je zatím samo}
Barva[1]:=1;     {zařazeno do skupiny 1}
Zarazena:=1;     {jedno město je zařazeno}
Konflikt:=false;
```



```
while (Vrchol>0) and not Konflikt do
  begin
  j:=Zasob[Vrchol];        {odebrat číslo ze zásobníku}
  Vrchol:=Vrchol-1;
  for i:=1 to N do
    if A[i,j] then         {vede silnice z j do i}
      if Barva[i]=0 then
        begin              {obarvit město i}
        Barva[i]:=-Barva[j];
        Vrchol:=Vrchol+1;
        Zasob[Vrchol]:=i;
        Zarazena:=Zarazena+1  {další město zařazeno}
        end
      else if Barva[i]=Barva[j] then
        Konflikt:=true;   {došlo ke kolizi - nelze rozdělit}
  if (Vrchol=0) and (not Konflikt) and (Zarazena<N) then
    begin
    {Zásobník je prázdný, ale ještě nebyla obarvena
     všechna města ani nedošlo ke konfliktu.
     Libovolné dosud nezařazené město můžeme zařadit
     do kterékoliv skupiny -> první dosud nezařazené
     dáme do skupiny 1}
    j:=1;
    while Barva[j]<>0 do j:=j+1;
    Zasob[1]:=j;               {nově zařazené město}
    Vrchol:=1;                 {v seznamu zařazených je samo}
    Barva[j]:=1;               {zařazeno do skupiny 1}
    Zarazena:=Zarazena+1;      {další město je zařazeno}
    end
  end;

{Vyhodnocení procesu rozdělování měst:}
if Konflikt then
  writeln('Rozdělení měst do dvou skupin neexistuje.')
else
  begin
  writeln('Rozdělení měst do dvou skupin je možné.');
  writeln('Jedno takové přípustné rozdělení vypadá takto:');
  write('1.skupina:');
  for i:=1 to N do
    if Barva[i]=1 then write(i:3);
```

```
  writeln;
  write('2.skupina:');
  for i:=1 to N do
    if Barva[i]=-1 then write(i:3);
  writeln
  end
end.
```

## CVIČENÍ

**1.** Navrhněte takovou modifikaci reprezentace grafu pomocí seznamu následníků jednotlivých vrcholů, která by byla vhodná pro případ ohodnocených grafů.

**2.** Orientovaný neohodnocený graf je na vstupu zadán počtem vrcholů a výčtem svých hran (uspořádaných dvojic čísel vrcholů, odkud kam hrana vede). Graf chceme v programu reprezentovat ve tvaru seznamu následníků jednotlivých vrcholů. Načtěte vstupní data a vytvořte z nich zvolenou vnitřní reprezentaci grafu.

**3.** Někdy mohou být v grafu ohodnoceny nejen hrany, ale i jeho vrcholy. Navrhněte vhodnou datovou reprezentaci neorientovaného ohodnoceného grafu, v němž je každá hrana ohodnocena dvěma celými čísly a každý vrchol jedním celým číslem.

**4.** Pravoúhlými souřadnicemi v rovině je zadána poloha $N$ letišť očíslovaných čísly od 1 do $N$. Je dáno výchozí letiště, ze kterého startuje letadlo, a dolet letadla, tj. maximální vzdálenost, jakou letadlo uletí bez mezipřistání. Určete, zda toto letadlo může doletět na všechna letiště.

**5.** Mezi $N$ městy označenými čísly od 1 do $N$ jezdí několik autobusových linek. Jsou dány trasy těchto linek jako seznamy čísel měst, v nichž jednotlivé linky staví. Všechny autobusové linky jsou provozovány v obou směrech po stejné trase.

a) Nalezněte a vypište všechna města, do nichž se můžeme pomocí autobusových linek dostat z města číslo 1 bez ohledu na počet přesedání.

b) Nalezněte a vypište všechna města, do nichž se můžeme pomocí autobusových linek dostat z města číslo 1 maximálně s dvěma přesedáními.

**6.** Mezi $N$ městy označenými čísly od 1 do $N$ je vybudována síť autobusových linek. Každá linka spojuje některou dvojici měst, autobusy staví



pouze na konečných stanicích. Mezi některými dvojicemi měst nemusí přímá autobusová linka existovat. O každé lince je známa cena jízdného (kladné číslo). Pro danou dvojici měst (výchozí a cílové) určete nejlevnější autobusové spojení z jednoho do druhého. Pokud existuje více různých tras s minimální cenou jízdného, vyberte z nich trasu s nejmenším počtem přesedání. Z více stejně dobrých možností zvolte jednu libovolnou.

**7.** Mezi $N$ městy označenými čísly od 1 do $N$ je vybudována síť autobusových linek. Každá linka spojuje některou dvojici měst, autobusy staví pouze na konečných stanicích. Mezi některými dvojicemi měst nemusí přímá autobusová linka existovat. O každé lince je známa doba jízdy autobusu a cena jízdného (kladná čísla). Pro danou dvojici měst (výchozí a cílové) určete nejlevnější autobusové spojení z jednoho do druhého. Pokud existuje více různých tras s minimální cenou jízdného, vyberte z nich tu, na níž je celková doba jízdy autobusem nejkratší. Jestliže existuje více takových tras se stejnou dobou jízdy, zvolte z nich trasu s nejmenším počtem přesedání. Pokud existuje více takových tras se stejným minimálním počtem přesedání, vyberte z nich trasu vedoucí městem číslo 2, je-li to možné. Z více stejně dobrých možností zvolte jednu libovolnou.

**8.** Pravoúhlými souřadnicemi v rovině je zadána poloha $N$ letišť očíslovaných čísly od 1 do $N$. Je znám dolet letadla, tj. maximální vzdálenost, jakou letadlo uletí bez mezipřistání. Je dáno výchozí letiště, z něhož letadlo startuje, a cílové letiště, kam letí.

a) Určete trasu letadla z výchozího na cílové letiště, na níž bude co nejmenší počet mezipřistání, bez ohledu na celkovou délku letu.

b) Nalezněte nejkratší cestu letadla z výchozího na cílové letiště, bez ohledu na počet mezipřistání.

c) Nalezněte nejkratší cestu letadla z výchozího na cílové letiště. Jestliže existuje více různých tras stejné minimální délky, vyberte z nich cestu s nejmenším počtem mezipřistání.

Z více stejně dobrých tras vyberte jednu libovolnou.

**9.** Z města $A$ do města $B$ vede silnice, na níž leží několik dalších měst. Vzájemné silniční vzdálenosti dvojic sousedních měst jsou dány. Mezi některými dvojicemi měst vedou autobusové linky. Každá linka má jen dvě zastávky — nástupní a cílovou. Pro každou linku je dána cena jízdného a v každém městě poplatek za přestup mezi linkami.

a) Zjistěte, zda je možné dojet z města $A$ do města $B$. Je povoleno libovolně přesedat mezi linkami a popř. v některých úsecích cesty použít i spoje jedoucí ve směru od města $B$ k městu $A$.

b) Existuje-li autobusové spojení z města $A$ do města $B$, nalezněte a vypište nejlacinější z nich (do ceny zahrňte vedle jízdného i poplatky za přestupování). Pokud existuje nejlacinějších spojení více, vyberte z nich takové, které vyžaduje co nejmenší počet přesedání. Jestliže i takových spojení existuje více různých, zvolte to, které je nejkratší z hlediska celkové vzdálenosti ujeté autobusy. Z více takových zvolte jedno libovolné.

**10.** Mezi $N$ městy označenými čísly od 1 do $N$ je vybudována silniční síť. Každá silnice spojuje vždy dvojici měst, mezi některými dvojicemi měst přímá silnice nevede. Všechny silnice jsou obousměrné a nekříží se nikde mimo města. Určete, zda je možné rozdělit města do dvou skupin tak, aby každá dvojice měst patřících do stejné skupiny byla spojena přímou silnicí (tzn. uvnitř každé skupiny vede přímá silnice mezi každými dvěma městy). Nezáleží přitom na velikosti jednotlivých skupin (jedna ze skupin může být případně i prázdná), ale každé město musí být do některé skupiny zařazeno. Je-li takové rozdělení měst do skupin možné, najděte ho. Z více možných řešení vypište jedno libovolné. Vstupem programu je počet měst $N$ a dále seznam všech silnic vedoucích mezi městy. Každá silnice je zadána dvojicí čísel měst, mezi nimiž vede.



# 10 ROZDĚL A PANUJ

S klasickou zásadou římských imperátorů „Rozděl a panuj" se v metodice programování setkáváme hned ve dvou různých významech. Někteří autoři tak označují samotný základní princip výstavby větších programových celků, který spočívá v rozdělení problému na několik dílčích relativně nezávislých částí a v samostatném naprogramování každé z nich formou podprogramu. Častěji však uvedené heslo označuje jednu konkrétní techniku návrhu algoritmů. Ta spočívá v rozdělení řešeného problému na dva nebo více menších problémů, které jsou svým charakterem podobné původnímu problému. Dílčí problémy pak vyřešíme nezávisle na sobě a z jejich výsledků složíme výsledek celkový. Pokud jsou vzniklé dílčí problémy stále ještě složité, řešíme je obdobně dalším dělením na stále menší problémy. Dělení končí ve chvíli, kdy k vyřešení zbývají malé snadno řešitelné problémy, které již zpracujeme přímo.

Programovací technika „rozděl a panuj" se hodí pouze pro určitý omezený okruh úloh, u nichž je rozdělení na menší nezávislé podúlohy stejného charakteru vůbec možné a přirozené. Je-li použitelná, vede zato zpravidla k poměrně krátkému a efektivnímu programu. Celá strategie má sama od sebe rekurzivní charakter a při její programové realizaci se také typicky používá rekurzivní volání podprogramů. Můžeme ji ale naprogramovat i bez použití rekurze, pokud mechanismus rekurze nahradíme v programu vlastním zásobníkem pro odkládání informací o dílčích podúlohách, které je ještě třeba zpracovat.

Metodu si předvedeme na několika konkrétních příkladech včetně jejího naprogramování s použitím rekurze a bez rekurze. Na technice „rozděl a panuj" jsou založeny dva známé a používané třídicí algoritmy — quicksort a třídění sléváním (mergesort). Oba patří k nejlepším algoritmům vnitřního třídění a vrátíme se k nim odkazem ještě v kap. 11, která je celá věnována problematice třídění. Již nyní nám ale oba algoritmy dobře poslouží jako ukázka programování technikou „rozděl a panuj". Třetím příkladem je elegantní rekurzivní řešení slavného problému hanojských věží. Konečně poslední ukázkový příklad předvádí, jak lze pomocí této metody vyhodnocovat aritmetické výrazy.

## 10.1 Quicksort

Typickým představitelem algoritmu založeného na principu rekurze a na technice „rozděl a panuj" je třídicí algoritmus quicksort. Patří k nejznámějším, nejlepším a nejpoužívanějším metodám vnitřního třídění. Popis quicksortu najdete v téměř každé učebnici programování. V naší odborné literatuře je však quicksort soustavně pronásledován tiskovými chybami, jeho programová realizace je uvedena špatně jak ve slovenském překladu knihy [19], tak třeba i v základní učebnici programování pro začátečníky [5]. Uvedeme ho proto i zde, a snad konečně bez chyby.

Celý algoritmus je založen na jednoduché základní myšlence. Zvolíme číslo $X$ (později se vrátíme k tomu, jak ho zvolit) a tříděné prvky přerovnáme v poli tak, aby v levé části pole byly pouze prvky menší nebo rovné $X$ a v pravé části prvky větší nebo rovné $X$. (Pozn.: Skutečnost, že prvky s hodnotou $X$ mohou být jak v levé, tak v pravé části pole, není podstatná, je jen technickou záležitostí usnadňující programovou realizaci algoritmu.) Po tomto rozdělení jistě platí, že prvky ležící v levé části pole zůstanou v této části i po úplném setřídění celého pole, a totéž platí i pro prvky v pravé části. Na pořadí prvků se stejnou hodnotou nezáleží, takže ani případný výskyt více prvků rovných $X$ (některé vlevo, jiné vpravo) nám nezpůsobí žádné problémy. Stačí tedy setřídit samostatně levý a pravý úsek pole. To jsou vlastně dvě dílčí menší úlohy naprosto stejného typu, jako byla úloha původní. Vyřešíme je proto stejným postupem. Postupné dělení pole na menší a menší úseky pokračuje tak dlouho, dokud nezískáme úseky délky 1. Ty jsou samy o sobě samozřejmě setříděné a nemusíme s nimi již nic dělat.

Popsaný algoritmus nyní ještě trochu upřesníme a doplníme. Nejprve se naučíme přerovnat prvky v poli do dvou úseků, vlevo menší a vpravo větší než zvolené $X$. K tomu nepotřebujeme nic jiného než dva pomocné indexy ukazující do právě zpracovávaného úseku pole. Index $I$ začíná na levém okraji úseku a zvětšuje se tak dlouho, dokud v poli nenajdeme první prvek větší než $X$. Ten nepatří do vytvářené levé části a má se přesunout někam vpravo. Podobně s indexem $J$ postupujeme od pravého okraje úseku směrem doleva tak dlouho, až poprvé narazíme na prvek menší než $X$. Ten se má zase přesunout do levé části. Oba nalezené prvky proto spolu vyměníme. Potom pokračujeme stejným způsobem v pohybu indexů $I$ a $J$ směrem ke středu a s výměnami prvků tak dlouho, dokud se oba indexy nesetkají. V tom okamžiku je celý úsek rozdělen na levou



část s menšími prvky a pravou část s většími prvky. Předěl obou úseků je dán polohou indexů $I$ a $J$. Jedno z možných naprogramování uvedeného postupu najdete v naší programové ukázce. Časová složitost tohoto dělení je lineární, počet operací je úměrný počtu prvků v úseku.

Dalším problémem, který musíme vyřešit, je volba hodnoty $X$, podle níž je úsek rozdělován. Rozhodně to nesmí být číslo menší než nejmenší prvek v rozdělovaném úseku, pak by vlastně k žádnému rozdělení nedošlo a při opakované stejné volbě $X$ by se výpočet dostal do nekonečného cyklu. Totéž platí pro čísla větší než největší prvek rozdělovaného úseku. Ostatní hodnoty $X$, tj. čísla z rozmezí od nejmenšího k největšímu prvku úseku, jsou přípustné a vedou vždy ke správnému výsledku. Na vhodné volbě čísla $X$ však závisí rychlost výpočtu. Nejlepší by bylo zvolit za číslo $X$ pokaždé tzv. **medián** právě zpracovávaného úseku. To je číslo s prostřední hodnotou ze všech čísel úseku (nikoli tedy jejich aritmetický průměr!). Například medián z dvaceti čísel je desáté největší z nich. Každý úsek by se v takovémto ideálním případě rozdělil vždy zhruba na dvě poloviny (až na nezbytné nesrovnalosti při dělení úseků liché délky a na možnost vzniku určitých rozdílů v délkách úseků v případě, že by se hodnota mediánu vyskytovala v děleném úseku vícekrát). Přibližně po $\log_2 N$ děleních bychom získali triviální úseky délky 1. Vzhledem k lineární složitosti dělení jednotlivých úseků má celý quicksort časovou složitost v nejlepším případě $O(N \log N)$, kde $N$ je počet tříděných prvků. Přesné vyhledání mediánu zkoumaného úseku je však dosti pracné a v quicksortu se nepoužívá.

V nejhorším případě bude výpočet probíhat značně pomaleji. Pokud za $X$ nešťastně zvolíme pokaždé nejmenší (nebo největší) prvek zpracovávaného úseku, rozdělí se v prvním kroku celé pole na dva úseky délek 1 a $N-1$, ve druhém kroku získáme z většího z nich dva úseky délek 1 a $N-2$ atd. Celkem se provede $N-1$ dělení, což vede k nepříznivé časové složitosti quicksortu v nejhorším případě $O(N^2)$. Zajímavé je ovšem zjištění, co můžeme očekávat v případě průměrném. S trochou vyšší matematiky se dá poměrně snadno odvodit, že průměrná časová složitost algoritmu je $O(N \log N)$. Jak uvidíme v kap. 11.2, je to nejlepší možná složitost algoritmu vnitřního třídění, jaké lze dosáhnout.

Vraťme se ale ještě k původní praktické otázce volby čísla $X$. V náhodně uspořádané posloupnosti vstupních dat na způsobu volby vlastně ani moc nezáleží. Nejjednodušší je vzít za $X$ vždy první prvek úseku.

Typicky se za $X$ volí například prvek ležící uprostřed zpracovávaného úseku nebo se $X$ počítá jako aritmetický průměr několika málo čísel z úseku (např. prvního, prostředního a posledního), aby se snížila pravděpodobnost té nejméně příznivé volby.

Vlastní programová realizace algoritmu již nevyžaduje téměř nic dalšího. Tříděná čísla jsou uložena v poli a veškeré třídění probíhá na místě v tomto jediném poli. Musíme již jenom rozhodnout, jakým způsobem v programu zajistíme zpracování dílčích úseků. Z hlediska zápisu programu je nejjednodušší využít možnosti rekurzivního volání procedury. Pokud bychom však rekurzi z jakéhokoli důvodu nechtěli nebo nemohli použít (například proto, že by v použitém programovacím jazyce nebyla povolena), nahradíme ji snadno dalším pomocným polem v roli zásobníku na ukládání krajních mezí těch úseků, které je ještě třeba setřídit.

Následují dvě ukázky, jak lze naprogramovat quicksort v jazyce Pascal, verze s použitím rekurze a verze bez rekurze. V obou případech je algoritmus zapsán v podobě procedury, která jako parametry obdrží identifikátor tříděného pole a počáteční a koncový index v poli. Tyto indexy určují, který úsek pole se má setřídit. Pro jednoduchost předpokládáme, že budeme třídit pole celých čísel.

Rekurzivní verze:

```
const MaxN = 100;          {maximální počet tříděných čísel}
type  Pole = array[1..MaxN] of integer;      {tříděná čísla}

procedure Quicksort1(var P:Pole; Zac,Kon:integer);
{setřídí v poli P úsek od indexu Zac do indexu Kon}
var X: integer;      {hodnota pro rozdělení na úseky}
    Q: integer;      {pomocné pro výměnu prvků v poli}
    I,J: integer;    {posouvané pracovní indexy v poli}
begin
  I:=Zac;
  J:=Kon;
  X:=P[(Zac+Kon) div 2];
    {za hodnotu X vezmeme pro jednoduchost
     prostřední prvek ve zkoumaném úseku}
  repeat
    while P[I] < X do I:=I+1;
    while P[J] > X do J:=J-1;
    if I < J then       {vyměnit prvky s indexy I a J}
```



```
      begin
      Q:=P[I]; P[I]:=P[J]; P[J]:=Q;
      I:=I+1; J:=J-1;    {posun indexů na další prvky}
      end
    else if I = J then
      {indexy I a J se sešly, oba dva ukazují na hodnotu X}
      begin
      I:=I+1; J:=J-1     {posun indexu na další prvky
                          - nutné kvůli ukončení cyklu}
      end
  until I > J;
  {úsek <Zac,Kon> je rozdělen na úseky <Zac,J> a <I,Kon>,
  které zpracujeme rekurzivním voláním procedury:}
  if Zac < J then Quicksort1(P,Zac,J);
  if I < Kon then Quicksort1(P,I,Kon);
end;  {procedure Quicksort1}
```

Nerekurzivní verze:

```
const MaxN = 100;          {maximální počet tříděných čísel}
      MaxNdiv2 = 50;      {= MaxN div 2 (velikost zásobníku}
type  Pole = array[1..MaxN] of integer;      {tříděná čísla}

var Zasob: array[1..MaxNdiv2] of
             record Zac,Kon: integer end;
                  {zásobník úseků čekajících na zpracování}
    Vrchol: 0..MaxN;                      {vrchol zásobníku}

procedure Quicksort2(var P:Pole; Zac,Kon:integer);
{setřídí v poli P úsek od indexu Zac do indexu Kon}
var X: integer;      {hodnota pro rozdělení na úseky}
    Q: integer;      {pomocné pro výměnu prvků v poli}
    Z,K: integer;    {začátek a konec zkoumaného úseku}
    I,J: integer;    {posouvané pracovní indexy v poli}
begin
  Zasob[1].Zac:=Zac;
  Zasob[1].Kon:=Kon;
  Vrchol:=1;          {celý tříděný úsek vložen do zásobníku}
  while Vrchol>0 do  {zásobník je neprázdný}
    begin
    Z:=Zasob[Vrchol].Zac;
```

```
    K:=Zasob[Vrchol].Kon;
    Vrchol:=Vrchol-1;  {odebrán jeden úsek ze zásobníku}
    I:=Z; J:=K;
    X:=P[(I+J) div 2];
    {za hodnotu X vezmeme pro jednoduchost
     prostřední prvek ve zkoumaném úseku}
    repeat
      while P[I] < X do I:=I+1;
      while P[J] > X do J:=J-1;
      if I < J then        {vyměnit prvky s indexy I a J}
        begin
        Q:=P[I]; P[I]:=P[J]; P[J]:=Q;
        I:=I+1; J:=J-1;    {posun indexů na další prvky}
        end
      else if I = J then
        {indexy I a J se sešly, oba dva ukazují na hodnotu X}
        begin
        I:=I+1; J:=J-1     {posun indexů na další prvky
                            - nutné kvůli ukončení cyklu}
        end
    until I > J;
    {úsek <Z,K> je rozdělen na úseky <Z,J> a <I,K>,
    které vložíme do zásobníku k dalšímu zpracování:}
    if Z < J then
      begin
      Vrchol:=Vrchol+1;
      Zasob[Vrchol].Zac:=Z;
      Zasob[Vrchol].Kon:=J
      end;
    if I < K then
      begin
      Vrchol:=Vrchol+1;
      Zasob[Vrchol].Zac:=I;
      Zasob[Vrchol].Kon:=K
      end
    end
end;  {procedure Quicksort2}
```

Zajímavé a poučné je zamyslet se nad tím, zda má nějaký význam pořadí, v němž ukládáme do zásobníku záznamy o obou úsecích vzniklých rozdělením jednoho zpracovávaného úseku pole. Z hlediska správnosti



výsledného utřídění na tomto pořadí rozhodně nezáleží. Každý záznam v zásobníku jednoznačně vymezuje jeden úsek pole, který je ještě třeba zpracovat. Tyto úseky jsou disjunktní a zcela nezávislé a není významné, která část pole bude utříděna dříve a která později. Přesto pro nás může být pořadí úseků v zásobníku dosti významné. Rozhodujícím způsobem totiž ovlivňuje paměťové nároky programu. Určuje nám, jak velký paměťový prostor musíme vyhradit pro zásobník.

V nejhorším případě by se nám mohlo stát, že by výpočet probíhal následovně. Po prvním rozdělení celého pole by se nejprve uložil na dno zásobníku záznam o úseku délky 2 a pak úsek délky $N-2$. Ten by byl vzápětí vyzvednut a rozdělen na dva úseky. Kratší z nich délky 2 by se uložil do zásobníku nejdříve, delší úsek délky $N-4$ potom. Stejným způsobem můžeme pokračovat ve sledování výpočtu dále. V jednom okamžiku se nám pak stane, že v zásobníku bude uloženo $N/2$ záznamů, z nichž každý odpovídá úseku pole délky 2. V tomto nejméně příznivém případě tedy jenom samotný zásobník spotřebuje paměť velikosti $N$ celých čísel. Takto velký prostor jsme museli pro zásobník předem vyhradit také v deklaracích našeho výše uvedeného programu.

Jestliže program doplníme o jeden jednoduchý test navíc, vystačíme s mnohem menším prostorem pro uložení zásobníku. Stačí porovnat vždy délky obou úseků, které vznikly rozdělením zpracovávaného úseku pole, a do zásobníku uložit nejdříve záznam odpovídající delšímu z nich. Záznam o kratším úseku se tedy pokaždé dostane na vrchol zásobníku a v dalším kroku výpočtu bude ihned ze zásobníku odstraněn a zpracován. Zásobník v tomto případě nejvíce poroste, pokud se budou všechny úseky dělit vždy na polovinu. Zkusme opět chvíli sledovat průběh takového výpočtu. Počáteční úsek $N$ čísel se rozdělí na dva úseky délky $N/2$ a záznamy o nich se uloží do zásobníku. Horní z nich ze zásobníku vyjmeme a po rozdělení úseku ho nahradíme dvěma záznamy o úsecích délky $N/4$. Vrchní záznam opět ze zásobníku odstraníme a nahradíme ho dvěma záznamy určujícími úseky délky $N/8$. Stejným způsobem pokračujeme tak dlouho, až získáme úseky délky 2. V tomto okamžiku dosáhne zaplnění zásobníku svého maxima. Zásobník obsahuje postupně ode dna k vrcholu záznamy určující úseky pole délky $N/2$, $N/4$, $N/8$, ..., 8, 4, 2, 2, tj. celkem přibližně $\log_2 N + 1$ záznamů. (Pozn.: Takto přesně budou délky úseků vycházet pouze pro ty hodnoty $N$, které jsou mocninou 2. Jinak je třeba uvažovat vždy horní nebo dolní celou část z výrazů $N/2$,

$N/4, \ldots, \log_2 N$. Na výsledné paměťové efektivitě algoritmu však tyto zaokrouhlovací nepřesnosti nic nezmění.)

Doplněním testu se nám tedy podařilo snížit paměťový prostor potřebný pro uložení zásobníku z $O(N)$ na $O(\log N)$. Pro velká $N$ to může znamenat dosti výraznou úsporu místa v paměti. Pokud bychom například třídili deset tisíc celých čísel a počítali bychom s uložením jednoho čísla do dvou bytů, pak původní řešení úlohy potřebuje vyhradit zásobník velikosti téměř 20 kB. Naproti tomu modifikace algoritmu s přednostním ukládáním delšího úseku do zásobníku vystačí se zásobníkem velikosti pouhých 60 bytů!

## 10.2 Třídění sléváním

Jiný třídicí algoritmus založený na principu postupného rekurzivního rozdělování tříděné posloupnosti na úseky se nazývá třídění sléváním (nebo také třídění slučováním), anglicky **mergesort**. Rovněž jeho základní myšlenka je velmi jednoduchá. Pole obsahující tříděná čísla rozdělíme na dvě poloviny a každou z nich samostatně setřídíme. Ze setříděných dílčích úseků pak sestavíme výslednou utříděnou posloupnost metodou „slévání" (upřesníme později). Setřídění obou dílčích úseků pole jsou úlohy stejného typu jako úloha původní. Vyřešíme je proto stejným postupem, což v programu zajistíme například rekurzivním voláním. Podobně jako u quicksortu pokračuje toto rozkládání na stále menší úseky tak dlouho, až získáme úseky délky 1, které jsou již samy o sobě setříděné a nic s nimi nemusíme dělat.

Slévání dvou setříděných úseků do jednoho se provede snadno, postup má zřejmě lineární časovou složitost. Oba slévané úseky se pomocí dvou pracovních indexů postupně procházejí a vždy menší z čísel se přemístí do vytvářené výsledné posloupnosti. Jedinou nevýhodou tohoto postupu je skutečnost, že potřebujeme ještě jedno pomocné pracovní pole, do kterého se výsledná setříděná posloupnost ukládá.

Třídění sléváním se tedy liší od quicksortu jenom tím, jakým způsobem provádíme rozdělení čísel na úseky a spojení setříděných dílčích úseků do jednoho. V quicksortu rozdělujeme čísla do úseků složitějším způsobem podle vybrané řídicí hodnoty, nemusíme zato už nic dělat se setříděnými dílčími úseky. U třídění sléváním provedeme rozdělení na dva úseky jednoduše na poloviny, setříděné úseky se pak ale ještě



musí pracně slévat. Díky dělení větších úseků na dílčí úseky poloviční délky je u třídění sléváním na rozdíl od quicksortu zajištěno, že rozdělení posloupnosti $N$ čísel až na elementární úseky délky 1 proběhne vždy v $\log_2 N$ krocích. Třídění sléváním má proto časovou složitost v nejlepším, nejhorším i průměrném případě $O(N \log N)$.

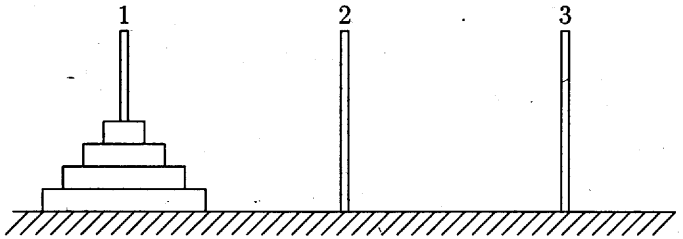
Ukážeme si nyní, jak vypadá algoritmus třídění sléváním naprogramovaný v Pascalu. Algoritmus je zapsán ve formě rekurzivní procedury, která jako parametry obdrží identifikátor tříděného pole, odkaz na jedno pomocné pole a počáteční a koncový index v tříděném poli. Indexy určují, který úsek pole se má setřídit. Pro jednoduchost předpokládáme, že budeme třídit pole $N$ celých čísel. Pomocné pole (parametr $Q$) slouží jako pracovní paměťový prostor pro dočasné uložení setříděného úseku během slévání. Jeho předávání v parametru procedury je v podstatě zbytečné, bylo by samozřejmě možné místo toho deklarovat takové pomocné pole jako lokální proměnnou procedury. To by sice umožnilo jednodušší způsob volání procedury (procedura by měla o jeden „zbytečný" parametr méně), ale ve svém důsledku by tato změna vedla k větším paměťovým nárokům programu (vzhledem k rekurzivnímu volání by se musel v paměti vyhradit prostor pro řadu exemplářů takového pole). Stejně jako v případě quicksortu by bylo možné nahradit mechanismus rekurzivního volání procedury vlastním zásobníkem naprogramovaným v Pascalu. Tuto nerekurzivní verzi zde již nebudeme uvádět a ponecháme ji do cvičení.

```
const MaxN = 100;          {maximální počet tříděných čísel}
type  Pole = array[1..MaxN] of integer;      {tříděná čísla}

procedure Mergesort(var P,Q:Pole; Zac,Kon:integer);
{setřídí v poli P úsek od indexu Zac do indexu Kon}
{Q je pomocné odkládací pole pro realizaci slévání}
var Stred:integer;     {index prostředku tříděného úseku}
    i,j,k: integer;    {pomocné indexy pro slévání}
begin
  Stred:=(Zac+Kon) div 2;
    {konec levého úseku při rozdělení úseku <Zac,Kon>}
  if Zac < Stred then Mergesort(P,Q,Zac,Stred);
    {utřídit levý úsek}
  if Stred+1 < Kon then Mergesort(P,Q,Stred+1,Kon);
```

```
    {utřídit pravý úsek}
  {slévání obou úseků:}
  i:=Zac;        {levý úsek}
  j:=Stred+1;    {pravý úsek}
  k:=Zac;        {výsledek}
  while (i<=Stred) and (j<=Kon) do
    {slévání setříděných úseků do pomocného pole Q}
    begin
    if P[i]<=P[j] then
      begin Q[k]:=P[i]; i:=i+1 end
    else
      begin Q[k]:=P[j]; j:=j+1 end;
    k:=k+1
    end;
  while i<=Stred do   {dokopírování zbytku levého úseku}
    begin Q[k]:=P[i]; i:=i+1; k:=k+1 end;
  while j<=Kon do     {dokopírování zbytku pravého úseku}
    begin Q[k]:=P[j]; j:=j+1; k:=k+1 end;
  {zbývá přenést setříděný úsek <Zac,Kon> zpět z Q do P:}
  for k:=Zac to Kon do
    P[k]:=Q[k]
end;  {procedure Mergesort}
```

### 10.3 Hanojské věže

Úloha o hanojských věžích patří dnes k neslavnějším a nejznámějším úlohám z oblasti „matematických rekreací" a programování. Máme tři




*Obr. 27* Hanojské věže pro $N = 4$



kolíky označené po řadě čísly 1, 2 a 3. Na kolíky budeme navlékat kotouče opatřené dírkou uprostřed. Máme celkem $N$ kotoučů, každý o jiném průměru. Na začátku hry je všech $N$ kotoučů nasazeno na kolíku 1, největší kotouč je zcela dole, na něm menší atd. až nejmenší kotouč je nahoře. Kolíky 2 a 3 jsou prázdné. Naším úkolem je přemístit všech $N$ kotoučů na kolík 2 tak, aby byly opět stejně uspořádány zdola nahoru od největšího k nejmenšímu. Kotouče musíme přenášet postupně jeden po druhém, k jejich odložení můžeme používat také kolík 3, ale nikdy nesmíme žádný kotouč odložit mimo kolíky. Navíc v každém okamžiku mezi dvěma přesuny kotoučů musí platit, že na žádném kolíku není nikde větší kotouč položen na menším.

Je to typická úloha na použití rekurze a metody výstavby programu stylem „rozděl a panuj". Snadnou úvahou zjistíme, že během přemisťování kotoučů z kolíku 1 na kolík 2 musíme nutně projít situací, že všech menších $N-1$ kotoučů je odloženo na kolík 3, na kolíku 1 zůstane pouze největší kotouč a kolík 2 je prázdný. Jenom v takovéto situaci můžeme přenést největší z kotoučů z kolíku 1 na kolík 2 a zahájit tím budování výsledné „věže z kotoučů" na kolíku 2. Celý proces přenesení věže $N$ kotoučů z kolíku 1 na kolík 2 se tím přirozeným způsobem rozkládá do tří kroků:

1. přenesení $N-1$ kotoučů z kolíku 1 na kolík 3
2. přenesení jednoho kotouče z kolíku 1 na kolík 2
3. přenesení $N-1$ kotoučů z kolíku 3 na kolík 2

Z těchto tří kroků představuje první a třetí vyřešení stejné úlohy, jako je úloha původní, ale s menším počtem kotoučů (a s jinými kolíky ve významu odkud kam se přenáší). Druhý z kroků je pak již elementární a můžeme ho ihned jednoduše provést. Kroky 1. a 3. vyřešíme stejným způsobem jako původní úlohu, opět rozkladem do podobných tří dílčích kroků. Stejně postupujeme dále, až se celá úloha rozloží do samých elementárních kroků.

Úlohu nyní vyřešíme v Pascalu. Program očekává na vstupu jedno číslo udávající celkový počet kotoučů $N$. Výše uvedeným postupem program pro toto $N$ určí způsob přenášení kotoučů a vytiskne ho ve tvaru uspořádané posloupnosti zpráv typu „přenes jeden kotouč z kolíku $X$ na kolík $Y$".

```
program Hanoj;
{Řešení úlohy o hanojských věžích}
var N: integer;     {počet kotoučů}

procedure Prenes(N,A,B,C: integer);
  {přenese věž N kotoučů z kolíku A na kolík B,
  přitom využívá kolík C jako pomocný pro odkládání}
  begin
  if N>0 then
    begin
    Prenes(N-1,A,C,B);
    writeln('Přenes kotouč z ', A, ' na ', B);
    Prenes(N-1,C,B,A)
    end
  end;  {procedure Prenes}

begin
write('Počet kotoučů: ');
readln(N);
Prenes(N,1,2,3)
end.
```

### 10.4 Vyhodnocení výrazu

Naším úkolem nyní bude vyčíslit zadaný aritmetický výraz. Problematice aritmetických výrazů je věnována celá dosti rozsáhlá kap. 12, ale již nyní si naznačíme jeden algoritmus, který využívá techniky „rozděl a panuj".

Pro jednoduchost budeme předpokládat, že aritmetický výraz je tvořen číselnými konstantami, binárními operátory +, −, *, / a kulatými závorkami s libovolnou možností vnoření. Je třeba dodržet pravidla pro vyhodnocování výrazů, která jsou běžná v matematice i v programovacích jazycích. To znamená, že pořadí vyhodnocování je určeno v první řadě závorkami, dále prioritami operátorů (násobení a dělení se provádí dříve než sčítání a odčítání) a konečně zásadou, že operátory téže priority se vyhodnocují zleva doprava.



Jak uvidíte později v kap. 12.2, úlohu vyhodnotit výraz je možné řešit mnoha velice rozdílnými postupy. My si nyní ukážeme pouze jednu metodu založenou na myšlence „rozděl a panuj". Ve výrazu nejprve určíme operátor, který bude při vyhodnocování proveden jako poslední. Tento operátor získáme poměrně snadno při jednom průchodu zápisem výrazu. Je to totiž operátor nejnižší priority stojící zcela mimo závorky ve výrazu co nejvíce vpravo. Celý vyhodnocovaný výraz se tímto operátorem rozdělí do dvou částí, na podvýraz stojící vlevo od vybraného operátoru a podvýraz vpravo od něj. Každý z těchto podvýrazů samostatně vyhodnotíme (stejným postupem, pomocí rekurzivního volání) a s jejich hodnotami pak provedeme poslední operaci určenou vybraným operátorem. Tím získáme výslednou hodnotu celého výrazu.

K podrobnějšímu popisu algoritmu včetně příkladu a návodu k řešení technických problémů se vrátíme ještě v kap. 12.2.

## CVIČENÍ

**1.** Upravte nerekurzivní verzi třídicího algoritmu quicksort uvedenou v kap. 10.1 tak, aby se maximálním možným způsobem zmenšila velikost zásobníku. Použijte k tomu metodu popsanou v závěru kap. 10.1. Jak velký zásobník budete v tomto konkrétním případě potřebovat, když $MaxN = 100$?

**2.** Má nějaký význam z hlediska věcné správnosti, časové nebo paměťové složitosti pořadí obou rekurzivních volání v případě rekurzivního naprogramování quicksortu? Dokázali byste nějak vylepšit proceduru *Quicksort1* z kap. 10.1?

**3.** Naprogramujte třídění sléváním bez použití rekurzivního volání procedury (rekurzivní volání nahraďte vlastním zásobníkem v programu).

**4.** Napište program na vyhodnocení jednoduchého aritmetického výrazu. Výraz může obsahovat pouze celočíselné konstanty bez znaménka, čtyři základní binární operátory +, −, *, / (poslední uvedený operátor představuje operaci celočíselného dělení) a kulaté závorky s neomezenou možností vnoření do sebe. Výraz je vyhodnocován podle pravidel obvyklých v matematice i v programovacích jazycích — nejprve podle závorek, pak podle priorit operátorů a operátory stejné priority zleva doprava. Vstupem programu je jeden syntakticky bezchybný aritmetický výraz, výsledkem výpočtu je hodnota zadaného výrazu (tj. jedno celé číslo). Využijte postup uvedený v kap. 10.4.

# 11 TŘÍDĚNÍ ÚDAJŮ

Pojem třídění se v programování používá trochu nepřesně pro označení činnosti, kterou bychom logicky asi spíše nazvali „uspořádání podle velikosti". Nepůjde nám totiž o žádné rozdělování čísel, záznamů nebo prvků do nějakých tříd, ale pouze o stanovení jejich pořadí. V novější české odborné literatuře bývá tato operace někdy přesněji a výstižněji označována jako řazení. Termín třídění je však natolik historicky vžitý a běžně používaný, že se ho zde budeme i nadále držet.

S problematikou třídění se každý z vás setkal v životě již mnohokrát. Podle abecedy jsou uspořádána jména žáků v seznamu třídy, jména telefonních účastníků v telefonním seznamu i cizí slova ve slovníku, podle získaného počtu bodů jsou seřazena mužstva v ligové tabulce i žáci při vyhodnocení přijímacího řízení na školu, v hodinách tělesné výchovy se řadíte podle své výšky, došlou poštu si můžete zakládat třeba podle data odeslání. Obecně vzato, tříděním rozumíme uspořádání množiny údajů vzestupně nebo sestupně podle zvolené klíčové položky. Údaje třídíme zpravidla proto, aby se nám s nimi lépe pracovalo. Třídění nám usnadňuje zejména pozdější vyhledávání uložených dat, jak jsme si ostatně již ukázali v kap. 6.

Třídicí algoritmy patří k nejznámějším standardním algoritmům vůbec a jsou velmi dobře propracovány. Využívají se v mnoha programech ze všech oblastí. V různých algoritmech třídění se uplatňuje řada rozmanitých programátorských postupů. Některé z nich jste již mohli poznat i v předcházejících kapitolách této knihy. Problematika třídění je podrobně zpracována v dostupné odborné literatuře, obvykle i s podrobnou analýzou efektivity jednotlivých algoritmů (např. [19], [12]). Nebudeme se jí zde proto věnovat příliš detailně a omezíme se pouze na získání základního nadhledu nad věcí.

V programech obvykle třídíme jednotlivé exempláře nějaké datové struktury typu pascalského záznamu. V takové struktuře bývá obsažena jedna význačná položka označovaná většinou jako **klíč**. Podle této klíčové položky se pak záznamy řadí do vzestupného nebo sestupného pořadí. Ostatní položky záznamu jsou z hlediska třídění nevýznamné, pouze se společně přemisťují. Pro zjednodušení výkladu i programových ukázek se proto můžeme omezit na třídění množiny celých čísel, která nám představují klíče tříděných datových záznamů. Čísla budeme uspořádávat



do vzestupného pořadí, tedy od nejmenšího k největšímu. Počet tříděných čísel $N$ je parametrem určujícím velikost konkrétní úlohy na třídění, v závislosti na $N$ budeme vyjadřovat časovou složitost jednotlivých algoritmů.

Metody třídění můžeme rozdělit do dvou hlavních skupin, a to na tzv. vnitřní třídění a vnější třídění. **Vnitřní třídění** se nazývá také třídění polí a použijeme ho tehdy, pokud velikost tříděné množiny dat umožňuje umístit všechna data najednou do vnitřní paměti počítače. Data jsou uložena v poli a algoritmy využívají možností přímého přístupu k jednotlivým prvkům. **Vnější třídění** slouží naopak pro případ, kdy je tříděných údajů příliš mnoho a do vnitřní paměti se nevejdou. Jsou proto umístěny v souborech na vnějších paměťových médiích a celé třídění je založeno na opakovaném čtení a vytváření souborů.

## 11.1 Algoritmy vnitřního třídění

Nejjednodušší používané algoritmy vnitřního třídění patří do skupiny tzv. **přímých metod**. Všechny mají několik společných rysů: algoritmus je velice krátký a jednoduchý, třídění probíhá na místě v poli dat bez dalších paměťových nároků, algoritmy mají kvadratickou časovou složitost. Právě poslední uvedená vlastnost vymezuje praktickou použitelnost přímých metod. Využívají se v případě, že tříděná data nejsou příliš rozsáhlá a časová složitost algoritmu $O(N^2)$ tedy není na závadu. Pro větší množiny dat slouží lepší třídicí algoritmy s časovou složitostí $O(N \log N)$. Ve stručnosti si nyní přiblížíme tři nejznámější přímé metody.

Třídění **přímým výběrem** spočívá v opakovaném vybírání nejmenšího čísla z dosud nesetříděných čísel. Je jen technickou záležitostí šikovně rozdělit pole s uloženými čísly na dva úseky: v levé části postupně vytváříme setříděnou posloupnost, vpravo zůstávají ještě nesetříděná čísla. V každém kroku výpočtu se vždy vybere nejmenší číslo z pravého úseku a připojí se na konec levého, již setříděného úseku čísel.

```
type Pole = array[1..N] of integer;

procedure PrimyVyber(var A: Pole);
{setřídění pole celých čísel A v rozsahu indexů od 1 do N
 metodou přímého výběru}
```

```
var i,j,k: integer;                {indexy prvků}
    X: integer;                    {pro výměnu prvků}
begin
  for i:=1 to N-1 do               {umístit číslo na pozici i}
    begin
    k:=i;
    for j:=i+1 to N do             {vyhledání minima}
      if A[j] <A[k] then k:=j;
    if k > i then                  {výměna prvků s indexy i, k}
      begin X:=A[k]; A[k]:=A[i]; A[i]:=X end
    end
end;  {procedure PrimyVyber}
```

Třídění **přímým vkládáním** (zatřiďováním) také rozděluje pole čísel na dva úseky. V levém úseku se vytváří uspořádaná posloupnost, vpravo zůstávají dosud nesetříděná čísla. Čísla z pravého úseku se berou jedno po druhém a do levého setříděného úseku se zatřiďují podle velikosti, kam patří.

```
type Pole = array[1..N] of integer;

procedure PrimeVkladani(var A: Pole);
{setřídění pole celých čísel A v rozsahu indexů od 1 do N
 metodou přímého vkládání}
var i,j: integer;                  {indexy prvků}
    X: integer;                    {pro výměnu prvků}
    Hledat: boolean;
begin
  for i:=2 to N do                 {zatřiďujeme číslo z pozice i}
    begin
    X:=A[i];
    j:=i-1;
    Hledat:=X<A[j];
    while Hledat do                {hledání správné pozice}
      begin
      A[j+1]:=A[j];
      j:=j-1;
      if j=0 then Hledat:=false else Hledat:=X<A[j]
      end;
    A[j+1]:=X
```



```
    end
end;  {procedure PrimeVkladani}
```

**Bublinkové třídění** (třídění výměnami) je založeno na trochu odlišném principu. Postupně se systematicky porovnávají dvojice sousedních prvků a vyměňují se spolu vždy, když menší číslo následuje po větším. Po ukončení výpočtu jsou všechny dvojice sousedních prvků dobře uspořádány, a proto je celé pole setříděné.

```
type Pole = array[1..N] of integer;

procedure BublinkoveTrideni(var A: Pole);
{setřídění pole celých čísel A v rozsahu indexů od 1 do N
 metodou bublinkového třídění}
var i,j: integer;                  {indexy prvků}
    X: integer;                    {pro výměnu prvků}
begin
  for i:=2 to N do
    for j:=N downto i do
      if A[j-1] > A[j] then        {vyměnit sousední prvky}
        begin X:=A[j-1]; A[j-1]:=A[j]; A[j]:=X end
end;  {procedure BublinkoveTrideni}
```

Na závěr přehledu přímých metod vnitřního třídění ještě poznamenejme, že poslední programová ukázka je sice názorná a přehledná, ale že to zdaleka není nejlepší možná programová realizace bublinkového třídění. Jednotlivé průchody polem je možné zkrátit vždy jen do místa poslední provedené výměny prvků, a tím i snížit celkový počet průchodů polem. Toto vylepšení si ale již necháme do cvičení.

Lepší třídicí algoritmy pracují v čase $O(N \log N)$. Konkrétní ukázky několika takových algoritmů jste mohli nalézt v předcházejících kapitolách knihy, a to včetně odvození jejich časové složitosti. **Třídění haldou** (podrobněji vysvětleno v kap. 6.7) je založeno na využití speciální datové struktury zvané halda. Z tříděných dat se nejprve vytvoří halda a z ní se pak postupně odebírá vždy nejmenší prvek. Výhodou algoritmu je jeho snadná realizace na místě v poli a časová složitost $O(N \log N)$ zaručená i v nejhorším případě. **Třídění sléváním** (kap. 10.2) spočívá v postupném spojování kratších setříděných úseků a vytváření delších. Také zaručuje

časovou složitost $O(N \log N)$ v nejhorším případě, má však dvojnásobné paměťové nároky pro uložení dat (potřebujeme ještě jedno pomocné pole stejné velikosti jako vstupní data pro slévání) a vyžaduje ještě další paměť pro realizaci rekurzivního rozkladu pole na úseky. K tomu slouží buď zásobník vytvořený přímo v programu, nebo systémový zásobník využívaný mechanismem rekurzivního volání procedur. **Quicksort** (viz kap. 10.1) je asi nejpoužívanější algoritmus vnitřního třídění. V nejhorším případě má sice časovou složitost $O(N^2)$, v průměru však dosahuje časové složitosti $O(N \log N)$ a mezi algoritmy této třídy je dokonce v průměrném případě nejrychlejší. Třídění dat probíhá v poli na místě, navíc potřebujeme pouze zásobník pro realizaci rekurzivního rozkladu pole na úseky (podobně jako u třídění sléváním).

Pro který třídicí algoritmus se tedy máme rozhodnout při psaní programu? Volba algoritmu závisí vždy na tom, co víme o zpracovávaných datech a jaké máme na program časové a paměťové nároky. Jestliže bude program třídit pouze malá data (o rozsahu řádově desítek nebo nejvýše stovek hodnot), je celkem zbytečné věnovat velké úsilí tomu, abychom sestrojili co nejlepší třídicí algoritmus. V takovém případě nám stejně dobře poslouží kterákoli přímá metoda. Pro zpracování rozsáhlých dat se ovšem vyplatí volit třídicí algoritmus s větší pečlivostí (jen si vzpomeňte na příklad s tříděním z kap. 3.3). V praxi se používají nejvíce quicksort a třídění haldou. První z nich zvolíme, když nám záleží na tom, aby program pracoval v průměru co nejrychleji, ale nevadí nám, jestliže pro některá vstupní data bude výpočet pomalejší. Haldové třídění je v průměru sice o něco pomalejší než quicksort, ale zato nám zajišťuje velmi rychlý výpočet pro jakákoli vstupní data. Do programu ho zařadíme zejména v situaci, kdy se výpočet nesmí za žádných okolností výrazněji zpozdit (například při zapojení počítače do řízení nebo zpracování dat v reálném čase).

## 11.2 Složitost vnitřního třídění

Vnitřní třídění patří k těm nemnoha netriviálním úlohám, u nichž umíme celkem jednoduše odvodit časovou **složitost problému**. Omezíme se na úlohu setřídit podle velikosti $N$ čísel, o nichž nemáme k dispozici žádné další informace. Pro jednoduchost budeme předpokládat, že třídíme $N$ navzájem různých čísel. Dokážeme, že nemůže existovat algoritmus,



který by tuto úlohu řešil v čase lepším než $O(N \log N)$. Tato mez se týká jak časové složitosti v nejhorším případě, tak dokonce i časové složitosti v průměrném případě. Algoritmy uvedené v druhé části kap. 11.1 jsou tedy z hlediska asymptotické časové složitosti optimální. Předpoklad, že o tříděných číslech nic nevíme, je pro další úvahy důležitý. Existuje totiž i třídicí algoritmus s lineární časovou složitostí (což je zdánlivě ve sporu s naším tvrzením), lze ho však použít pouze pro třídění celých čísel z předem známého ne příliš velkého rozmezí, aby jimi bylo možné přímo indexovat pole.

Jestliže o tříděných číslech nic nevíme (mohou to být například reálná čísla), nemůžeme je v programu použít ani k indexování polí, ani v nějakých řídicích konstrukcích typu přepínače, příkazu větvení case apod. Celý výpočet musí být řízen pouze pomocí operací typu porovnání dvou tříděných čísel. Můžeme si představit strom všech možných výpočtů jistého třídicího algoritmu. Kořen stromu představuje začátek výpočtu, listy stromu odpovídají bodům ukončení výpočtu pro jednotlivá vstupní data. Vnitřní uzly stromu představují provedení podmíněných příkazů v programu, kdy se další výpočet větví do dvou možných variant pokračování na základě výsledku porovnání dvou čísel. Jde tedy o binární strom, jehož všechny vnitřní uzly mají oba dva následníky (výpočet vždy nějak pokračuje, ať je výsledek porovnání jakýkoli).

Každou $N$-tici různých čísel můžeme uspořádat přesně $N!$ způsoby (= počet permutací $N$-prvkové množiny). Symbol $N!$ znamená faktoriál čísla $N$ a je roven součinu všech celých čísel od 1 do $N$. Existuje tedy v principu $N!$ různých vstupních dat velikosti $N$, která je možné třídicímu algoritmu předložit. Průběh výpočtu závisí právě jen na uspořádání vstupní $N$-tice čísel, nikoli na konkrétních hodnotách. Každou z $N!$ permutací dat musí algoritmus správně setřídit a pokaždé dojít ke stejné výsledné posloupnosti. To znamená, že pro každá z těchto dat se musí výpočet někde odlišit od výpočtu pro jiná data. Náš uvažovaný strom všech možných cest výpočtu třídicího algoritmu bude mít proto přesně $N!$ listů. Hloubka listu ve stromu (tj. údaj, na kolikáté hladině stromu list leží) je rovna počtu porovnání provedených během výpočtu pro odpovídající vstupní data. Maximum z těchto hloubek je rovno výšce celého stromu a určuje časovou složitost algoritmu v nejhorším případě. Pokusíme se najít odhad, jaká musí tato výška být.

Binární strom s pevně zvoleným počtem listů $L$ má nejmenší výšku

tehdy, jsou-li všechny listy stromu na téže hladině nebo ve dvou sousedních hladinách (jinými slovy: hloubky libovolných dvou listů se liší nejvýše o 1). Jeho výška je potom rovna přibližně $\log_2 L$. Pro řádový odhad složitosti můžeme zanedbat řešení takových detailů, jako zda jde o horní nebo dolní celou část uvedeného logaritmu apod. V našem případě uvažujeme strom s $N!$ listy, a ten má tedy výšku minimálně $\log_2(N!)$.

K důkazu, že problém vnitřního třídění má časovou složitost v nejhorším případě $O(N \log N)$, již zbývá provést pouze malou formální matematickou úpravu. Lze dokázat, že pro vhodnou konstantu $C$ platí nerovnost

$$\log_2(N!) \geqq CN \log_2 N,$$

takže výšku uvažovaného stromu všech výpočtů můžeme zdola odhadnout také výrazem $CN \log_2 N$. Tím budeme s celým důkazem hotovi. Chybí nám již jenom odvození poslední uvedené nerovnosti. K tomu jsou ovšem zapotřebí poněkud vyšší matematické znalosti. Nerovnost můžeme odvodit například na základě tzv. Stirlingova vzorce, který aproximuje rychlost růstu funkce $N!$ právě pomocí exponenciální funkce. Ze Stirlingova vzorce plyne nerovnost $N! \geqq (N/\mathrm{e})^N$ a odtud již zlogaritmováním dostaneme hledanou nerovnost.

Na závěr si ještě ukážeme, co se změní, budeme-li se zajímat o časovou složitost problému vnitřního třídění v průměrném případě. Musíme tedy vyjít od aritmetického průměru spočítaného z hloubek všech $N!$ listů v našem stromě a opět hledat dolní odhad této hodnoty. Dá se snadno ukázat, že pro pevně zvolený počet listů nabývá tento výraz minima opět v případě, že jsou všechny listy stromu na téže hladině nebo ve dvou sousedních hladinách. Aritmetický průměr hloubek všech listů je pak přibližně roven výšce stromu. Může být menší než výška stromu, ale méně než o 1. Odhadneme ho proto zdola výrazem $\log_2(N!) - 1$. Po stejné formální úpravě jako v předchozím případě odtud již přímo plyne, že časová složitost vnitřního třídění je i v průměrném případě $O(N \log N)$.



## 11.3 Přihrádkové třídění

Právě jsme dokázali, že obecně není možné setřídit $N$ záznamů rychleji než v čase $O(N \log N)$. V některých speciálních případech však můžeme použít třídicí algoritmus s lineární časovou složitostí. Tento zdánlivý spor ve skutečnosti žádným rozporem není. Třídicí algoritmus s časovou složitostí $O(N)$ skutečně existuje, nelze ho však použít univerzálně, ale pouze pro třídění celých čísel (resp. záznamů, kde klíčem je celé číslo) z předem známého, ne příliš velkého rozmezí. Algoritmus se nazývá přihrádkové třídění nebo také **třídění rozdělováním**, v anglicky psané literatuře ho najdete nejčastěji pod označením **radixsort**.

Zmíněný dosti neurčitý požadavek, aby tříděná čísla byla všechna z „ne příliš velkého rozmezí", je způsoben tím, že k práci použijeme pole indexované přímo tříděnými čísly. Maximální přípustný rozsah tříděných hodnot proto závisí na tom, jak velké pole si můžeme dovolit vymezit v paměti počítače pro tento účel. Typicky to bývají řádově tisíce.

Předpokládejme nyní, že známe dvě konstanty $D, H$ takové, že všechna tříděná čísla (klíče tříděných záznamů) jsou z intervalu $\langle D, H \rangle$. Rozdíl $H - D$ přitom není příliš velký. Připravíme si $H - D + 1$ „přihrádek", které označíme hodnotami $D, D+1, \ldots, H-1, H$. Přihrádky jsou na začátku výpočtu prázdné. Postupně budeme zpracovávat tříděné záznamy a každý z nich zařadíme vždy do té přihrádky, jejíž označení se shoduje s klíčem záznamu. Záznamy zařazené do téže přihrádky mají stejný klíč a na jejich vzájemném pořadí v principu nezáleží. Jestliže však budeme přihrádky realizovat jako seznamy a budeme v nich zachovávat pořadí jednotlivých záznamů, docílíme toho, že algoritmus bude tzv. **stabilní** (tzn. v setříděné posloupnosti bude zachováno původní vzájemné pořadí záznamů se stejným klíčem). Po rozmístění všech záznamů do přihrádek pak už jenom stačí obsah přihrádek vypsat jednu po druhé.

Popsaný algoritmus je možné naprogramovat různými způsoby. Jednou možností by bylo ukládat obsah jednotlivých přihrádek v podobě lineárních spojových seznamů tříděných záznamů. Vstupem do celé datové struktury pak bude pole ukazatelů na tyto seznamy. Toto pole bude indexováno od $D$ do $H$. Při psaní programu se bez problémů obejdeme i bez dynamických datových struktur. Třídění ale nebude probíhat na místě, kromě pole $A[1..N]$ obsahujícího tříděné záznamy budeme potřebovat druhé pole $B[1..N]$, do kterého se bude postupně ukládat setříděná posloupnost. Pomyslné přihrádky tedy budou vlastně úseky v poli $B$.

Prvky pomocného pole $C[D..H]$ nám poslouží nejprve jako počítadla velikosti jednotlivých přihrádek. Příslušné hodnoty snadno získáme při jednom průchodu polem $A$. Po určení velikosti všech přihrádek přetransformujeme hodnoty uložené v poli $C$ tak, aby určovaly polohu jednotlivých přihrádek v poli $B$. Přepočet zvolíme tak, aby prvek $C[K]$ udával index posledního místa v poli $B$, kde má být uložen záznam s klíčem $K$. Potom již stačí projít ještě jednou polem $A$ a jednotlivé záznamy přímo umisťovat na základě jejich klíče na správné výsledné pozice do pole $B$. Hodnoty prvků pole $C$ se přitom patřičně zmenšují tak, aby v každém okamžiku $C[K]$ určovalo index toho místa v poli $B$, kam má být vložen další záznam s klíčem $K$. Pokud budeme procházet polem $A$ od konce, bude celý algoritmus třídění stabilní (záznamy se stejným klíčem se do přihrádky umisťují vždy od konce přihrádky).

Přihrádkové třídění má časovou složitost $O(N + (H - D))$. Všechny popsané kroky jsou jednoduché lineární průchody některým z polí, někdy polem $A$ velikosti $N$, někdy polem $C$ velikosti $H - D$. Algoritmus je tedy skutečně lineární, ale měřeno nejen v $N$, nýbrž také v $(H - D)$. To znamená, že v případě malého počtu tříděných záznamů, jejichž klíče mohou být z velkého rozmezí $H - D$, bude algoritmus z časového hlediska nevýhodný. Rozdíl $H - D$ může být vyšší než například $N \log N$.

Procedura v Pascalu ukazuje přihrádkové třídění zapsané pomocí polí. Pole tříděných záznamů je předáváno parametrem $A$, který je vstupní a zároveň výstupní.

```
const N = 1000;                    {počet záznamů}
      D = 1;                       {dolní mez klíče}
      H = 100;                     {horní mez klíče}

type Pole = array[1..N] of record
                             klic: integer;
                             {cosi dalšího {...}}
                           end;

procedure PrihradkoveTrideni(var A: Pole);
var B: Pole;                       {setříděné záznamy}
    C: array[D..H] of integer;     {hranice přihrádek}
    I, K: integer;
begin
```



```
  {Velikosti přihrádek:}
  for I := D to H do
    C[I] := 0;
  for I := 1 to N do
    begin
    K := A[I].klic;
    C[K] := C[K] + 1
    end;
  {Konce přihrádek:}
  for I := D+1 to H do
    C[I] := C[I] + C[I-1];
  {Zařazení záznamů do přihrádek:}
  for I := N downto 1 do
    begin
    K := A[I].klic;
    B[C[K]] := A[I];
    C[K] := C[K] -1
    end;
  {Předání setříděného pole ven:}
  A := B
end;  {procedure PrihradkoveTrideni}
```

Přihrádkové třídění lze použít i v situaci, jestliže je rozsah hodnot klíčů příliš velký. Je-li například klíčem až šesticiferné číslo, nebudeme moci deklarovat pole $C$ o milionu prvcích. Klíč proto rozdělíme na skupiny cifer vhodné velikosti a použijeme víceprůchodové přihrádkové třídění. Průchodů bude tolik, na kolik částí klíč rozdělíme. V našem případě bychom pravděpodobně rozdělili klíč na první trojčíslí a druhé trojčíslí. Přihrádkové třídění podle trojciferného klíče nám již nebude působit žádné problémy, pole velikosti 1 000 se do paměti počítače zpravidla vejde. Celé třídění bude dvouprůchodové. Můžeme samozřejmě zvolit i jiné rozdělení klíče, třeba na tři dvojčíslí. Vystačíme pak s pracovním polem velikosti pouze 100, ale musíme zato při třídění provést tři průchody.

Víceprůchodové přihrádkové třídění je možné organizovat dvěma způsoby. První možností je rozdělit nejprve všechny záznamy do přihrádek podle části klíče z nejvyšších řádů, každou z přihrádek pak samostatně rozdělit na menší přihrádky podle další části klíče atd., dokud nebudou záznamy zcela uspořádány podle nejnižší části klíče. Z hlediska zápisu programu je lepší opačná cesta, a sice od úseku klíče

nejnižších řádů k řádům nejvyšším. Tento zdánlivě nepřirozený postup je korektní díky tomu, že je přihrádkové třídění stabilní. Nejprve utřídíme posloupnost záznamů podle části klíče tvořené nejnižšími řády, potom celou posloupnost najednou utřídíme podle vyšších řádů klíče atd., až k řádům nejvyšším. Každý průchod přihrádkového třídění zachovává pořadí záznamů pocházející z předchozích průchodů všude tam, kde se právě sledované úseky klíče shodují. Na závěr posledního průchodu budou proto záznamy správně seřazeny podle celého klíče.

Algoritmus má časovou složitost $O(P(N+(H-D)))$, kde $P$ je počet průchodů a $(H - D)$ velikost rozmezí přípustných hodnot části klíče určené pro jeden průchod. Jde totiž přesně o $P$-krát provedený algoritmus jednoprůchodového přihrádkového třídění.

### 11.4 Hledání $K$-tého nejmenšího prvku

Vnitřnímu třídění je velmi blízká úloha nalézt $K$-tý nejmenší prvek z dané množiny dat pro zadané číslo $K$ z rozmezí od 1 do $N$ ($N$ je počet prvků). Speciálním případem této úlohy je určení tzv. **mediánu**, což je prostřední prvek z množiny (tedy případ $K = N/2$). Dejte pozor na to, že $K$-tý nejmenší prvek nemusí představovat $K$-tou nejmenší hodnotu obsaženou v množině, nevylučuje se totiž opakovaní prvků téže hodnoty mezi zkoumanými daty.

Úlohu by bylo možné samozřejmě řešit tak, že bychom všechna čísla setřídili a vzali pak $K$-té v pořadí. To ale nebude nejlepší postup, jak $K$-té nejmenší číslo nalézt, neboť tříděním všech čísel jsme vykonali mnoho zbytečné práce. Existuje více lepších algoritmů pro řešení této úlohy. Nejlepší z nich je poměrně komplikovaný, dosahuje však lineární časové složitosti i v nejhorším případě. My si zde ukážeme jiné, mnohem jednodušší a používanější řešení. Algoritmus je založen na podobném principu jako quicksort, má také kvadratickou časovou složitost v nejhorším případě, v průměru však pracuje v lineárním čase.

Pokud se chcete s následujícím algoritmem seznámit podrobněji, přečtěte si dříve výklad quicksortu v kap. 10.1. Zde si vysvětlíme pouze odlišnosti. Stejně jako u quicksortu zvolíme vhodnou hodnotu $X$ a podle ní přerovnáme čísla v poli do dvou úseků — vlevo menší nebo rovna $X$, vpravo větší nebo rovna $X$. Porovnáme počet prvků levého úseku s hodnotou $K$ (hledáme $K$-té nejmenší číslo) a na základě tohoto porovnání



snadno zjistíme, ve kterém z úseků se hledané číslo nachází. Libovolný prvek levého úseku je totiž menší nebo roven libovolnému prvku pravého úseku, takže je-li například $K$ menší než velikost levého úseku, můžeme nadále pracovat pouze s levým úsekem pole a pravého si vůbec nemusíme všímat. Na rozdíl od quicksortu se tedy v dalším kroku výpočtu bude pracovat vždy jen s jedním z obou úseků vzniklých dělením. Díky tomu není zapotřebí používat rekurzi, snadno ji nahradíme cyklem.

Výsledný algoritmus si nyní ukážeme naprogramovaný ve tvaru funkce v Pascalu.

```
type  Pole = array[1..MaxN] of integer;      {uložení čísel}

function NaleztKty(var P:Pole; Zac,Kon,K:integer):integer;
{nalezení K-tého nejmenšího prvku metodou založenou
 na modifikaci třídicího algoritmu Quicksort:
 v poli celých čísel P v úseku od indexu Zac do indexu Kon
 vyhledá K-té nejmenší číslo a vrátí ho jako funkční hodnotu}
var X: integer;      {hodnota pro rozdělení na úseky}
    Q: integer;      {pomocné pro výměnu prvků v poli}
    I,J: integer;    {posouvané pracovní indexy v poli}

begin
  while Zac < Kon do
    begin
    X:=P[K];         {jedna možná volba, lze i jinak}
    I:=Zac;
    J:=Kon;
    repeat
      while P[I] < X do I:=I+1;
      while P[J] > X do J:=J-1;
      if I < J then       {vyměnit prvky s indexy I a J}
        begin
        Q:=P[I]; P[I]:=P[J]; P[J]:=Q;
        I:=I+1; J:=J-1;   {posun indexů na další prvky}
        end
      else if I = J then
        {indexy I a J se sešly, oba dva ukazují na hodnotu X}
        begin
        I:=I+1; J:=J-1    {posun indexů na další prvky
                            - nutné kvůli ukončení cyklu}
```

```
    end
  until I > J;
  {úsek <Zac,Kon> je rozdělen na úseky <Zac,J> a <I,Kon>}
  if K < I then Kon:=J; {dál budeme hledat v levém úseku}
  if K > J then Zac:=I; {dál budeme hledat v pravém úseku}
  end;
  NaleztKty:=P[K]
end;  {function NaleztKty}
```

## 11.5 Vnější třídění

Algoritmy vnějšího třídění slouží k uspořádání rozsáhlých souborů dat. Řeší situaci, kdy se všechna data nevejdou najednou do operační paměti počítače, a k jejich utřídění proto nelze použít žádnou metodu vnitřního třídění. Jen si představte, že byste měli za úkol zpracovat například výsledky parlamentních voleb nebo podklady získané při sčítání lidu.

Podstatou vnějšího třídění je přesouvání tříděných údajů mezi sekvenčními soubory dat uloženými na vnějších paměťových médiích (dnes nejčastěji na magnetických discích, dříve se pro tento účel využívaly zejména magnetické pásky). Při tomto přesouvání se data třídí po částech, postupně se z nich vytvářejí delší a delší setříděné úseky (označované někdy jako monotonie, běhy nebo sledy), až budou nakonec všechna data setříděna v jednom souboru. Vnější paměti mají dostatečnou kapacitu i pro uložení velmi rozsáhlých množin dat. Při práci se sekvenčními soubory zato musíme respektovat jedno zásadní omezení oproti práci s poli, a to skutečnost, že v každém okamžiku je přístupný pouze jediný prvek souboru a že soubor je vždy čten postupně od začátku do konce. Jen na okraj poznamenejme, že toto omezení neplatí při práci s diskovými soubory s přímým přístupem, u nichž může být přímo zpracováván libovolný prvek.

Základní operací, kterou provádíme se setříděnými úseky dat, je tzv. **slučování** (nebo také „slévání") monotonií. Jde v principu o stejný postup, s jakým jsme se seznámili již dříve v algoritmu vnitřního třídění slučováním (kap. 10.2). Rozdíl spočívá jen v tom, že slučované úseky nyní nejsou uloženy v poli, nýbrž jsou v souborech. I v případě slučování úseků uložených v poli byly tyto úseky procházeny pouze postupně zleva



doprava, tedy přesně tak, jak to vyžaduje přístup k datům v sekvenčních souborech. Při vnějším třídění bude vlastní slučování setříděných úseků probíhat naprosto stejně. Musíme však nějak odlišně zorganizovat celkový postup třídění. V případě souborů totiž nelze rozumně použít rekurzivní rozklad shora dolů vždy na úseky poloviční délky, jako tomu bylo u třídění polí. Budeme muset postupovat naopak zdola nahoru, vytvářet nejprve kratší úseky a z nich slučováním úseky delší.

Existuje řada různých algoritmů vnějšího třídění. Ukážeme si alespoň základní myšlenky některých z nich. Vůbec nejjednodušší postup se nazývá **přímé slučování**. Přímé slučování je zbytečně pomalé, a proto se prakticky nepoužívá, snadno si však na něm vysvětlíme princip celého třídění i možné cesty, jak lze efektivitu vnějšího třídění zvyšovat. Předpokládejme, že na začátku práce jsou všechna tříděná data uložena v jednom souboru. Neznáme jejich počet a nevíme ani nic o jejich počátečním uspořádání.

Nejprimitivnější verze přímého slučování postupuje následovně. Ze všech tříděných čísel vytvoříme nejprve uspořádané dvojice, z nich pak slučováním vzniknou čtveřice, z nich osmice atd., až budou všechna čísla setříděna. Není-li počet všech čísel zrovna dělitelný dvěma, čtyřmi, osmi atd., nic se neděje, poslední vytvářená „$k$-tice" bude prostě kratší. Vyřešení tohoto problému je jen okrajovou technickou záležitostí a nebudeme se jím více zabývat. Vytvářené uspořádané $k$-tice čísel budeme ukládat opět do souboru, jednoduše jednu za druhou. Obecně tedy v $i$-tém kroku výpočtu zpracováváme soubor čísel tvořený posloupností uspořádaných úseků délky $2^{i-1}$ a vytváříme z něj jiný soubor, který bude obsahovat přesně tatáž čísla, ale seřazená do posloupnosti uspořádaných úseků délky $2^i$. Výchozí soubor pak můžeme smazat. Výpočet začíná prvním krokem se vstupním souborem obsahujícím setříděné „úseky" délky 1 — každé číslo samo tvoří jeden takový primitivní setříděný úsek. Celé třídění skončí ve chvíli, kdy všech $N$ čísel vytvoří jediný uspořádaný úsek. K tomu je zapotřebí vykonat přesně takový počet kroků $P$, pro který platí $2^{P-1} < N$ a $2^P \geq N$. Provedený počet kroků $P$ je tedy roven horní celé části z $\log_2 N$.

Zbývá ukázat, jak se jeden krok výpočtu vlastně provede. V nejjednodušší podobě popisované metody se každý krok skládá ze dvou fází (fází nazýváme jeden průchod všemi tříděnými čísly). V první, tzv. **rozdělovací fázi** rozdělíme setříděné úseky ze vstupního souboru do dvou pomocných

souborů $P1$ a $P2$. Do každého z nich překopírujeme polovinu úseků ze vstupního souboru. Protože předem nevíme, kolik úseků vstupní soubor obsahuje, provádíme rozdělování tak, že čteme vstupní soubor a jednotlivé uspořádané úseky zapisujeme střídavě do $P1$ a do $P2$. Je-li počet všech úseků lichý, bude $P2$ obsahovat o jeden úsek méně než $P1$. Ve druhé, tzv. **slučovací fázi** pak ze souborů $P1$ a $P2$ vytváříme výsledný výstupní soubor. Nejprve sloučíme první úseky ze souborů $P1$ a $P2$ do jednoho úseku dvojnásobné délky a ten zapíšeme do výstupního souboru, potom totéž uděláme s druhými úseky souborů $P1$ a $P2$ atd. až do konce souborů. Pokud obsahuje $P1$ o jeden úsek více než $P2$, překopírujeme tento úsek nakonec do výstupního souboru.

Pro stanovení časové složitosti algoritmu je třeba uvážit, které elementární operace jsou časově nejnáročnější, a máme je tedy použít k výpočtu. U vnějšího třídění jsou z hlediska časových nároků nejvýznamnější vstupní a výstupní operace, tj. čtení dat ze souboru a zápis do souboru. Doba porovnání dvou čísel je oproti nim zanedbatelná, všechny operace prováděné ve vnitřní paměti počítače jsou o několik řádů rychlejší než přístup k datům umístěným na vnějším paměťovém médiu. Bude nás proto zajímat, kolikrát algoritmus přečte a zapíše číslo do souboru. V průběhu třídění je každé číslo vždy přečteno z jednoho souboru a poté zapsáno do jiného souboru. Počet čtení a zápisů je tedy pokaždé stejný, takže pro porovnání různých metod vnějšího třídění stačí uvažovat operace čtení.

Výše popsaný algoritmus dvoufázového přímého slučování probíhá v $\log_2 N$ krocích, v každém z nich je každé tříděné číslo dvakrát čteno ze souboru. Celkem se tedy vykoná $2N \log_2 N$ čtení.

Uvedený základní algoritmus vnějšího třídění můžeme snadno zrychlit hned třemi různými způsoby, které lze navíc libovolně kombinovat, a tím výsledný efekt ještě násobit: sloučit obě fáze jednoho kroku výpočtu do fáze jediné, zvýšit stupeň slučování a nezačínat od elementárních setříděných úseků délky 1, ale od větších. Všechny tři úpravy nyní popíšeme podrobněji.

První úpravou je nahrazení dvoufázového slučování **slučováním jednofázovým**. Samostatná rozdělovací fáze je totiž naprosto zbytečná, z hlediska rozložení čísel do setříděných úseků nepřináší vůbec nic nového. Přitom nám spotřebuje přibližně stejně času jako užitečná fáze slučovací. Nabízí se proto myšlenka spojit vždy slučovací fázi jednoho kroku výpočtu s rozdělovací fází kroku následujícího a tím zdvojnásobit rychlost výpočtu. Spojení obou fází je přitom velmi snadné, stačí neukládat při slučování úseků všechny vytvořené delší setříděné úseky do jediného výstupního souboru, ale hned je rozdělovat střídavě do dvou souborů. Samostatná rozdělovací fáze tak zůstane zachována pouze jednou, na samém začátku třídění. Počet kroků výpočtu se touto úpravou nezměnil, v každém kroku (kromě prvního) však bude každé z tříděných čísel načítáno pouze jednou. Vykoná se tedy pouze přibližně $N \log_2 N$ operací čtení, tzn. zhruba polovina ve srovnání s dvoufázovým slučováním.

Dalšího zrychlení výpočtu můžeme dosáhnout zvýšením **stupně slučování**. Jestliže budeme rozdělovat utříděné úseky ne do dvou, ale třeba do čtyř souborů, a potom slučovat vždy čtyři úseky do jednoho výsledného, získáme v $i$-tém kroku výpočtu setříděné úseky délky $4^i$. Pro celkový počet kroků $P$ pak platí $4^{P-1} < N$ a $4^P \geq N$, tedy $P$ je rovno přibližně $\log_4 N$. Obecně se při slučování stupně $S$ při výpočtu vykoná přibližně $\log_S N$ kroků. Na základě pravidel pro počítání s logaritmy si snadno odvodíte, že například $\log_4 N = \frac{1}{2} \log_2 N$ nebo $\log_8 N = \frac{1}{3} \log_2 N$. Zvýšíme-li tedy stupeň slučování ze 2 na 4, zrychlíme výpočet dvakrát, při stupni slučování 8 je zrychlení trojnásobné.

Někdo by nyní mohl přijít s nápadem stále zvyšovat stupeň slučování, a tím třídění libovolně zrychlovat. Celá záležitost ale není tak jednoduchá. Při zvýšení stupně slučování ze 2 na 4 se rychlost výpočtu sice zdvojnásobila, ale další zrychlování výpočtu postupuje již mnohem pomaleji, příslušná závislost je logaritmická. Poměr zrychlení výpočtu při slučování stupně $S$ oproti stupni 2 je totiž určen podílem $(\log_S N)/(\log_2 N) = \log_2 S$. K desetinásobnému zrychlení vnějšího třídění by tedy bylo zapotřebí použít slučování stupně $2^{10} = 1\,024$. Zde však narazíme hned na dva problémy. První z nich není tak podstatný, ale je třeba vzít ho také v úvahu. Slučování takto vysokého stupně přestává mít zanedbatelné časové nároky po stránce operací prováděných s čísly ve vnitřní paměti. Vybrání menšího ze dvou čísel je zanedbatelně krátké ve srovnání s přečtením čísla ze souboru. Čas potřebný pro výběr nejmenšího z 1 000 čísel však již v tomto srovnání zanedbatelný není (můžeme si ovšem trochu pomoci například použitím haldy — viz kap. 6.7). Celkové zrychlení výpočtu bude tedy ještě nižší, než jsme očekávali. Druhý problém spojený se záměrem použít slučování příliš vysokého stupně může být zásadnějšího rázu. Pokusíte-li se totiž například otevřít zároveň

1 000 souborů, s velkou pravděpodobností narazíte na zásadní odpor ze strany operačního systému vašeho počítače. Operační systém musí poskytovat každému otevřenému souboru jistou pracovní vyrovnávací paměť a s ohledem na omezenou velikost paměti omezuje také počet souborů, které mohou být zároveň otevřeny (typicky na několik desítek souborů). Pro dvoufázové slučování stupně $S$ potřebujete pracovat najednou s $S+$ $+1$ soubory (nejprve se vždy jeden soubory rozdělí na $S$ a těchto $S$ se pak slučuje do jednoho), v případě jednofázového slučování stupně $S$ je zapotřebí $2S$ souborů ($S$ souborů je vždy vstupních a $S$ výstupních). Ještě v nedávné době se rozsáhlejší data ukládala převážně na magnetické pásky. Maximální možný stupeň slučování byl v takovém případě omezen ještě výrazněji, a to počtem (fungujících) magnetopáskových jednotek připojených k počítači. Pokud měl počítač například 8 zařízení pro snímání magnetické pásky, bylo možné použít dvoufázové slučování stupně nejvýše 7 nebo jednofázové slučování stupně 4.

Poslední metoda zvyšování rychlosti vnějšího třídění spočívá v práci s delšími elementárními setříděnými úseky, které vstupují do prvního kroku slučování. U přímého slučování se vycházelo vždy od základu od „úseků" délky 1. V každé vstupní posloupnosti čísel (s výjimkou málo pravděpodobného případu, že by všechna čísla byla srovnána sestupně) jsou však samy od sebe obsaženy jisté delší rostoucí úseky. Bylo by škoda nevyužít je, když nás jejich příprava vůbec nic nestojí. Často jsou vstupní data dokonce již částečně předtříděna a rostoucí úseky v nich mohou být i dosti dlouhé. Při třídění budeme postupovat stejně jako dosud, ale v prvním kroku výpočtu budeme pracovat s již existujícími rostoucími úseky ve vstupní posloupnosti dat. Je-li těchto úseků $X$, sníží se počet kroků výpočtu z $\log_2 N$ na $\log_2 X$ a celková časová složitost z $O(N \log N)$ na $O(N \log X)$. Této metodě se říká **přirozené slučování**.

Ve stejné úvaze však můžeme pokročit ještě o kousek dál. Proč spoléhat na náhodu a čekat, kolika a jak dlouhými rostoucími úseky je vstupní posloupnost dat tvořena, když si v tomto směru můžeme snadno pomoci sami a data si do rostoucích úseků jednoduše **předtřídit**. Celkové množství tříděných dat nám sice nedovoluje umístit je najednou do paměti a utřídit je nějakou metodou vnitřního třídění, můžeme to však provést postupně vždy s takovou částí dat, jaká se nám do vnitřní paměti vejde. Výsledkem tohoto „nultého kroku" bude uspořádání všech tříděných čísel do relativně velkých rostoucích úseků. Tento krok navíc



spojíme s dosud samostatnou rozdělovací fází prvního kroku slučování, takže se nám dokonce ani nezvýší počet čtení dat ze souboru. Budou-li mít předtříděné úseky délku řádově několika tisíc až několika desítek tisíc čísel (což je reálný odhad), pak při stupni slučování 2 tímto krokem vlastně „přeskočíme" asi 10 až 15 prvních kroků přímého slučování. Obecně při délce předtříděných úseků $M$ je počet těchto úseků $X = N/M$ a celková časová složitost třídění vychází $O(N \log(N/M))$.

Z dosud uvedených variant si můžete vybrat do vašeho vlastního algoritmu vnějšího třídění to, co sami uznáte za vhodné. Rozumnou volbou je například jednofázové přirozené slučování stupně 4 nebo třeba jednofázové slučování stupně 3 s předtříděním počátečních úseků nějakou metodou vnitřního třídění. Existují i další, komplikovanější metody vnějšího třídění, které dosahují ještě vyšší rychlosti. K nejznámějším patří polyfázové třídění a kaskádové třídění, o kterých se můžete dočíst například v knize [19].

Na závěr výkladu věnovaného metodám vnějšího třídění si ukážeme alespoň jednu ze zmíněných metod zapsanou v Pascalu. Pro jednoduchost zvolíme metodu poměrně jednoduchou, a sice základní jednofázové přirozené slučování stupně 2. Třídit budeme soubor celých čísel nazvaný `VSTUP.DAT`, výsledný utříděný soubor bude pojmenován `VYSTUP.DAT`. Předpokládejme, že oba tyto soubory jsou textové. Dočasné pracovní soubory, pomocí nichž třídíme, je však pro urychlení výpočtu lepší realizovat jako datové soubory. Vzhledem k nezbytnosti pracovat v programu se soubory musíme se v této ukázce poněkud odchýlit od zásady nepoužívat v knize odlišnosti Turbo Pascalu od normy. Pro manipulaci se soubory použijeme některé procedury Turbo Pascalu, které v normě jazyka Pascal nejsou obsaženy (assign — přiřazení fyzického jména identifikátoru souboru, close — uzavření souboru, erase — smazání souboru).

```
program VnejsiTrideni;
{Třídění rozsáhlého souboru celých čísel metodou
jednofázového přirozeného slučování stupně 2.
Vstupem programu je textový soubor VSTUP.DAT s čísly,
výstupem bude utříděný textový soubor VYSTUP.DAT.}

type Soubor = file of integer;      {tvar pracovních souborů}
var Vstup, Vystup: text;   {vstupní a výstupní soubor čísel}
    A1, A2, B1, B2: Soubor;                {pracovní soubory}
```

```
    Cislo, Stare: integer;              {pro rozdělení na úseky}
    Zapis1: boolean;    {příznak zápisu do souboru A1 vs. A2}
    Slevat: boolean;          {příznak, že je ještě co slévat}
    KonecA: boolean;    {příznak, že je výsledek v A1 vs. B1}

procedure Slucovani(var X1, X2, Y1, Y2: Soubor;
                    var Slevat: boolean);
{vlastní algoritmus jedné fáze jednofázového přirozeného
slučování stupně 2 ze souboru X1, X2 do souboru Y1, Y2.
V parametru Slevat vrací informaci, zda je ještě třeba
pokračovat v slučování}

var C1: integer;                                {číslo z X1}
    C2: integer;                                {číslo z X2}
    Stare: integer;           {předchozí hodnota X1 nebo X2}
    Pokr1, Pokr2: boolean;     {pokračovat číst úsek z X1,2}
    Zapis1: boolean;   {příznak zápisu do souboru Y1 vs. Y2}
    Eof1, Eof2: boolean;     {soubor X1,2 zpracován do konce}

begin  {procedure Slucovani}
  reset(X1);
  reset(X2);
  rewrite(Y1);
  rewrite(Y2);
  Eof1 := eof(X1);
  Eof2 := eof(X2);
  read(X1, C1);
  read(X2, C2);
  Zapis1 := true;                   {začneme zapisovat do Y1}
  Slevat := false;                     {zatím není co slévat}
  while not (Eof1 or Eof2) do     {slučovat uspořádané úseky}
    begin
    Pokr1 := true;                    {můžeme pokračovat z X1}
    Pokr2 := true;                    {můžeme pokračovat z X2}
    while Pokr1 and Pokr2 do
      if C1 < C2 then
        begin
        if Zapis1 then write(Y1, C1)
                  else write(Y2, C1);
        if eof(X1) then
          begin
```



```
          Pokr1 := false;
          Eof1 := true
          end
        else
          begin
          Stare := C1;
          read(X1, C1);
          if C1 < Stare then Pokr1 := false
          end
        end
      else
        begin
        if Zapis1 then write(Y1, C2)
                  else write(Y2, C2);
        if eof(X2) then
          begin
          Pokr2 := false;
          Eof2 := true
          end
        else
          begin
          Stare := C2;
          read(X2, C2);
          if C2 < Stare then Pokr2 := false
          end
        end;
    while Pokr1 do              {dokopírovat konec úseku z X1}
      begin
      if Zapis1 then write(Y1, C1)
                else write(Y2, C1);
      if eof(X1) then
        begin
        Pokr1 := false;
        Eof1 := true
        end
      else
        begin
        Stare := C1;
        read(X1, C1);
        if C1 < Stare then Pokr1 := false
        end
```

```
      end;
    while Pokr2 do                    {dokopírovat konec úseku z X2}
      begin
      if Zapis1 then write(Y1, C2)
                else write(Y2, C2);
      if eof(X2) then
        begin
        Pokr2 := false;
        Eof2 := true
        end
      else
        begin
        Stare := C2;
        read(X2, C2);
        if C2 < Stare then Pokr2 := false
        end
      end;
    Zapis1 := not Zapis1;
    if Zapis1 then Slevat := true    {exist. aspoň dva úseky}
    end;
  if not Eof1 then      {ještě přepsat poslední úsek(y) z X1}
    begin
    if Zapis1 then
      write(Y1, C1)
    else
      begin
      write(Y2, C1);
      Slevat := true
      end;
    while not eof(X1) do
      begin
      Stare := C1;
      read(X1, C1);
      if C1 < Stare then
        Zapis1 := not Zapis1;
      if Zapis1 then write(Y1, C1)
                else write(Y2, C1);
      end
    end;
  if not Eof2 then      {ještě přepsat poslední úsek(y) z X2}
    begin
```



```
    if Zapis1 then
      write(Y1, C2)
    else
      begin
      write(Y2, C2);
      Slevat := true
      end;
    while not eof(X2) do
      begin
      Stare := C2;
      read(X2, C2);
      if C2 < Stare then
        Zapis1 := not Zapis1;
      if Zapis1 then write(Y1, C2)
                else write(Y2, C2);
      end
    end;
  close(X1);
  close(X2);
  close(Y1);
  close(Y2);
end;  {procedure Slucovani}

procedure Prepis(var F: Soubor);
{přepsání utříděných čísel z datového souboru F
do výsledného textového souboru Vystup}
var Cislo: integer;
begin
  reset(F);
  rewrite(Vystup);
  while not eof(F) do
    begin
    read(F, Cislo);
    write(Vystup, Cislo:6)
    end;
  close(F);
  close(Vystup);
end;  {procedure Prepis}

begin
{Přiřazení souborů:}
```

```
  assign(Vstup, 'VSTUP.DAT');
  assign(A1, 'A1.$$$');
  assign(A2, 'A2.$$$');
  assign(B1, 'B1.$$$');
  assign(B2, 'B2.$$$');
  assign(Vystup, 'VYSTUP.DAT');

  {Počáteční rozdělení úseků čísel ze Vstup do A1, A2:}
  reset(Vstup);
  rewrite(A1);
  rewrite(A2);
  Stare := -maxint;
  Zapis1 := true;                    {začneme zapisovat do A1}
  Slevat := false;                      {zatím není co slévat}
  while not eof(Vstup) do
    begin
    read(Vstup, Cislo);
    if Cislo < Stare then
      begin
      Zapis1 := not Zapis1;     {další úsek do druhého souboru}
      Slevat := true                    {jsou alespoň dva úseky}
      end;
    if Zapis1 then
      write(A1, Cislo)
    else
      write(A2, Cislo);
    Stare := Cislo
    end;
  close(Vstup);
  close(A1);
  close(A2);
  KonecA := true;

  {Vlastní slučování mezi soubory A1,'A2 a B1, B2:}
  while Slevat do
    begin
    Slucovani(A1, A2, B1, B2, Slevat);
    KonecA := false;              {utříděná data jsou v B1, B2}
    if Slevat then
      begin
      Slucovani(B1, B2, A1, A2, Slevat);
```



```
      KonecA := true              {utříděná data jsou v A1, A2}
      end
    end;  {while Slevat}

  {Závěrečné přepsání setříděných čísel
   do výstupního textového souboru:}
  if KonecA then
    Prepis(A1)              {výsledek je v souboru A1}
  else
    Prepis(B1);             {výsledek je v souboru B1}

  {Zrušení pracovních souborů:}
  erase(A1);
  erase(A2);
  erase(B1);
  erase(B2);
  end.
```

## CVIČENÍ

**1.** Vylepšete programovou realizaci bublinkového třídění podle návodu v kap. 11.1 (zkrácení průchodů polem do místa poslední výměny prvků).

**2.** Naprogramujte třídění haldou s využitím procedur pro práci s haldou, které jsou uvedeny v kap. 6.7.

**3.** Napište proceduru na třídění haldou, která dostane již naplněné pole tříděných čísel, čísla utřídí na místě v tomto poli (bez použití dalšího pomocného pole!) a setříděnou posloupnost čísel v poli zanechá.

**4.** Naprogramujte víceprůchodové přihrádkové třídění. Předpokládejte, že klíčem tříděných záznamů je nejvýše osmiciferné celé číslo.

**5.** Upravte program z kap. 11.5 tak, aby jednofázové přirozené slučování mělo stupeň 4 a aby se třídily několikasložkové záznamy podle zvoleného klíče.

**6.** Naprogramujte některou metodu vnějšího třídění, která bude využívat předtřídění počátečních uspořádaných úseků ve vnitřní paměti.

# 12 ZPRACOVÁNÍ ARITMETICKÝCH VÝRAZŮ

K běžným problémům, které musíme při psaní programů řešit, patří formální úpravy a zpracování různých výrazů. Tuto problematiku si přiblížíme na práci s jednoduchými aritmetickými výrazy. Využijeme při tom mnoha znalostí, které jsme získali v předchozích kapitolách knihy — práci se zásobníkem, procházení stromem do hloubky, metodu „rozděl a panuj".

V celé kapitole budeme pracovat s běžnými aritmetickými výrazy, které jsou tvořeny číselnými konstantami, binárními operátory $+, -, *$ a $/$ a kulatými závorkami $(,)$. Pro jednoduchost nebudeme ve výrazech připouštět výskyt proměnných, všechny konstanty budou celočíselné a také operátor $/$ budeme chápat jako znak celočíselného dělení. Doplnění výrazů o proměnné, přechod k reálné aritmetice a případné přidání dalších operátorů a funkcí je jen technickou záležitostí a nic nemění na principu zde uvedených algoritmů a postupů.

## 12.1 Reprezentace aritmetického výrazu

Z matematiky i z programování jsme všichni zvyklí na běžný zápis aritmetických výrazů, který bývá někdy označován jako **infixová notace**. Každý binární operátor je ve výrazu umístěn mezi operandy, s nimiž se má provést příslušná operace. Pořadí, v jakém jsou jednotlivé operátory prováděny, je určeno v první řadě pomocí závorek. Výraz může obsahovat libovolné množství párů závorek, není ani omezena úroveň vnořování závorek do sebe. Operátory stojící vedle sebe na stejné závorkové úrovni se vyhodnocují v pořadí určeném jejich prioritami a operátory téže priority zleva doprava. Násobení a dělení má přitom vyšší prioritu než sčítání a odčítání.

**Příklad**

Výraz $(65 - 3 * 5)/(2 + 3)$ má hodnotu 10. Při jeho vyhodnocení se díky vyšší prioritě spočítá nejprve součin a pak rozdíl v levé závorce. Potom se spočítá součet v pravé závorce a nakonec se oba mezivýsledky podělí.

Vedle běžně používané infixové notace se zejména v programování objevují ještě další dvě formy zápisu aritmetických výrazů nazývané post-



fix a prefix. S tzv. **postfixovou notací** jste se mohli setkat například u některých starších typů kapesních kalkulaček. Dříve se pro ni užívalo také označení reversní polská logika. Spočívá v tom, že každý operátor v zápisu výrazu následuje bezprostředně za oběma operandy, k nimž se vztahuje. Přitom každým z jeho operandů může být buď přímo číselná konstanta, nebo nějaký podvýraz stejné postfixové struktury. U **prefixové notace** je situace přesně opačná, každý operátor je umístěn v zápisu výrazu před svými dvěma operandy. Obě tyto notace mají společnou jednu velmi důležitou vlastnost. Vůbec neobsahují a nepotřebují závorky, pořadí vyhodnocování jednotlivých operátorů je plně určeno jejich umístěním ve výrazu.

**Příklad**

Výraz z předchozího příkladu

$$(65 - 3 * 5)/(2 + 3)$$

má v postfixové notaci podobu

$$65\ 3\ 5\ *\ -2\ 3\ +\ /$$

a v prefixové notaci ho zapíšeme jako

$$/\ -\ 65\ *\ 3\ 5\ +\ 2\ 3.$$

Postfixová i prefixová notace výrazu jsou na první pohled pro člověka dosti nepřehledné. Jsou však velmi výhodné pro jakékoli manipulace s výrazy v programech. Jak uvidíte v kap. 12.2, vyhodnocení aritmetického výrazu zapsaného v prefixové nebo postfixové notaci je ve srovnání s vyhodnocením běžného infixového zápisu mnohem jednodušší a rychlejší. Všimněte si ještě, že všechny tři formy zápisu téhož aritmetického výrazu mají stejné pořadí číselných konstant. Liší se pouze pořadím a umístěním znamének a v případě infixu ještě přidáním závorek.

Jestliže chceme v programu pracovat s aritmetickými výrazy, musíme si zvolit nějakou jejich vnitřní reprezentaci. Výraz si můžeme například uložit jako znakový řetězec v některé z uvedených notací. Jinou užívanou podobou je reprezentace aritmetického výrazu **binárním stromem**. V uzlech stromu budou uloženy jednotlivé elementy výrazu. Vnitřní

uzly stromu obsahují operátory, každý z listů představuje jednu číselnou konstantu. Strom reprezentující výraz neobsahuje žádné závorky, pořadí vyhodnocení je plně určeno strukturou stromu. Binární operátor se vztahuje vždy k těm dvěma operandům, které jsou reprezentovány levým a pravým následníkem příslušného uzlu s operátorem. Každým z těchto následníků je buď list stromu obsahující přímo číselnou konstantu, nebo netriviální podstrom reprezentující podvýraz.

Ukážeme si, jak je možné deklarovat uzly binárního stromu pro reprezentaci aritmetického výrazu v jazyce Pascal. Vnitřní uzly stromu i listy jsou nutně stejného typu, aby z nich bylo možné vytvořit společnou stromovou strukturu. Přitom každý vnitřní uzel musí obsahovat položku typu char pro uložení operátoru, zatímco v listech potřebujeme položku typu integer pro uložení číselné hodnoty. Nejjednodušší je umístit do každého uzlu obě dvě tyto položky a využívat vždy jen jednu z nich. Snadno poznáme, která z nich obsahuje platnou hodnotu. Stačí otestovat, zda ukazatele $L$ a $P$ uložené v uzlu mají hodnotu `nil`. Pokud ano, jde o list obsahující číselnou hodnotu, v opačném případě jsme ve vnitřním uzlu stromu s operátorem.

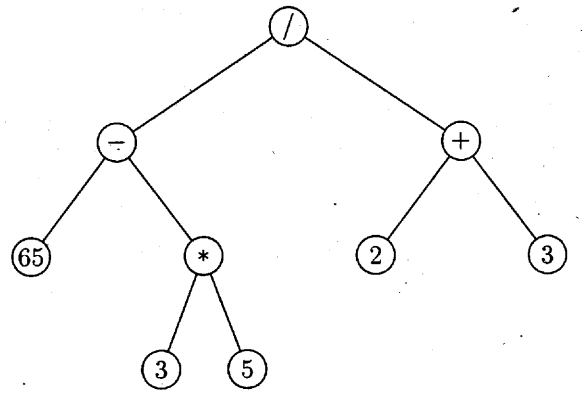
```
type Uk   = ^Uzel;
     Uzel = record
              Hodnota: integer;  {číselná konstanta - list}
              Znak: char;        {binární operátor}
              L,P : Uk           {levý a pravý následník}
            end;
```

Jinou možností by bylo použít pro uzly binárního stromu variantní záznam jazyka Pascal. Do každého uzlu umístíme pouze údaje, které v něm skutečně potřebujeme a které v něm mají smysl, navíc tam ale přibude jedna logická položka rozlišující, zda jde o list. Všimněte si, že v tomto případě listy nemusí vůbec obsahovat ukazatele na následníky.

```
type Uk = ^Uzel;
     Uzel = record
            case List: boolean of
              true: (Hodnota: integer);
              false:(Znak: char;
                     L,P : Uk)
            end;
```



Jednotlivé notace aritmetického výrazu (prefixová, infixová a postfixová) úzce souvisejí se způsoby procházení binárního stromu do hloubky




*Obr. 28* Výraz z předchozího příkladu $(65 - 3 * 5)/(2 + 3)$ reprezentovaný binárním stromem

(preorder, inorder, postorder — viz kap. 8.1). Pokud budeme procházet strom reprezentující aritmetický výraz metodou **preorder** a jako akci prováděnou v uzlech pokaždé vypíšeme obsah uzlu (tzn. položku *Znak* u vnitřních uzlů, položku *Hodnota* u listů), dostaneme prefixový zápis příslušného výrazu. Podobně při průchodu metodou **postorder** obdržíme postfixovou notaci. Trochu složitější je situace v případě průchodu stromem metodou **inorder**. Jestliže budeme opět pouze vypisovat obsah jednotlivých uzlů, získáme sice zápis výrazu v infixové notaci, ale bez závorek. Takový zápis je samozřejmě zcela bezcenný, neboť z hlediska vyhodnocování nemusí odpovídat původnímu výrazu, neodráží se v něm stavba výrazu zachycená původně ve struktuře procházeného stromu. Chceme-li získat z binárního stromu správný zápis výrazu v infixové notaci, musíme průchod inorder spojený s vypisováním obsahu jednotlivých uzlů doplnit ještě o vypsání levé, resp. pravé závorky vždy při prvním vstupu do uzlu, resp. před posledním opuštěním uzlu.

Všechny tři postupy sloužící k vytvoření zápisu aritmetického výrazu v prefixové, postfixové a infixové notaci na základě vnitřní reprezentace

výrazu binárním stromem si nyní ukážeme naprogramované v Pascalu. Jednotlivé procedury očekávají ve svém vstupním parametru $K$ odkaz na kořen stromu. Vytvořený výraz pro jednoduchost vypisují přímo na výstup.

```
procedure Prefix(K: Uk);
  {průchod binárním stromem s kořenem K metodou preorder}
  {vypsání aritmetického výrazu v prefixové notaci}
begin
if K^.L = nil then          {list}
  write(K^.Hodnota,' ')
else                        {vnitřní uzel}
  begin
  write(K^.Znak);
  Prefix(K^.L);
  Prefix(K^.P);
  end
end;

procedure Postfix(K: Uk);
  {průchod binárním stromem s kořenem K metodou postorder}
  {vypsání aritmetického výrazu v postfixové notaci}
begin
if K^.L = nil then          {list}
  write(K^.Hodnota,' ')
else                        {vnitřní uzel}
  begin
  Postfix(K^.L);
  Postfix(K^.P);
  write(K^.Znak);
  end
end;

procedure Infix(K: Uk);
  {průchod binárním stromem s kořenem K metodou inorder}
  {vypsání aritmetického výrazu v infixové notaci}
begin
if K^.L = nil then          {list}
  write(K^.Hodnota)
else                        {vnitřní uzel}
```



```
  begin
  write('(');
  Infix(K^.L);
  write(K^.Znak);
  Infix(K^.P);
  write(')');
  end
end;
```

## 12.2 Vyhodnocení aritmetického výrazu

Nejdůležitější akcí prováděnou s aritmetickými výrazy je jejich vyhodnocování. Ukážeme si, jak lze vyhodnotit výraz uložený ve vnitřní reprezentaci binárním stromem a jak se vyhodnocuje výraz zapsaný v prefixovém, postfixovém a infixovém tvaru.

Výraz reprezentovaný binárním stromem by bylo možné vyhodnotit tak, že bychom ve stromě postupně vyhledávali okamžitě vyhodnotitelné podvýrazy, vyhodnocovali je a nahrazovali je jejich spočtenými hodnotami. Znamenalo by to vyhledávat vždy ty uzly stromu, jejichž oba následníci jsou listy. Takovouto trojici uzlů snadno nahradíme uzlem jediným — listem obsahujícím vypočtenou hodnotu dílčího podvýrazu. Tímto postupem se postupně snižuje počet uzlů stromu, až nakonec zbyde z celého stromu uzel jediný. Kořen stromu nebude mít již žádné následníky a bude obsahovat hodnotu celého původního výrazu. Popsaný algoritmus je zcela korektní a vede vždy ke správnému výsledku. Vyhledávání podvýrazů ve stromě je však dosti složité a zbytečně pomalé.

Aritmetický výraz uložený ve tvaru binárního stromu vyhodnotíme velmi snadno jiným postupem. Použijeme rekurzivní algoritmus ve stylu metody „rozděl a panuj“ (viz kap. 10). V kořeni stromu je uložen znak operace, která se má provést jako poslední. Její operandy jsou reprezentovány levým a pravým podstromem kořene. Stačí tedy vyhodnotit oba podstromy a se získanými výsledky pak provést poslední operaci. Je-li podstrom listem, má svoji hodnotu přímo uloženu v sobě. Jinak k vyhodnocení podstromu použijeme rekurzivní volání téhož vyhodnocovacího algoritmu.

Následující rekurzivní funkce zapsaná v Pascalu realizuje právě popsaný postup vyhodnocování. Ve vstupním parametru $K$ očekává ukaza-

tel na kořen stromu reprezentujícího aritmetický výraz, funkční hodnotou je vyčíslená hodnota tohoto výrazu.

```
function Vyhodnoceni(K: Uk): integer;
  {vyhodnocení aritmet. výrazu reprezentovaného stromem}
  {K - kořen stromu}
begin
with K^ do
  begin
  if L = nil then
    Vyhodnoceni := Hodnota
  else
    case Znak of
    '+': Vyhodnoceni := Vyhodnoceni(L) + Vyhodnoceni(P);
    '-': Vyhodnoceni := Vyhodnoceni(L) - Vyhodnoceni(P);
    '*': Vyhodnoceni := Vyhodnoceni(L) * Vyhodnoceni(P);
    '/': Vyhodnoceni := Vyhodnoceni(L) div Vyhodnoceni(P);
    end
  end
end;  {function Vyhodnoceni}
```

Vyhodnocení výrazu zapsaného v postfixové notaci je také velmi snadné. Použijeme jednoduchý algoritmus s lineární časovou složitostí (měřeno délkou zpracovávaného výrazu, tj. počtem jeho operandů a operátorů). Algoritmus využívá jeden pomocný zásobník k ukládání číselných hodnot — mezivýsledků. Postfixový výraz čteme zleva doprava prvek po prvku (jedním prvkem výrazu budeme rozumět jeden operand nebo operátor). Operand, tj. číselnou konstantu, vždy rovnou uložíme do zásobníku. Přečteme-li binární operátor, vyjmeme ze zásobníku vrchní dvě čísla, provedeme s nimi operaci určenou právě přečteným znaménkem a výsledek opět uložíme do zásobníku. Pozor musíme dát pouze při provádění nekomutativních operací odčítání a dělení. Číslo z vrcholu zásobníku je pravým operandem, číslo vybírané ze zásobníku jako druhé je levým operandem. Pokud byl zpracovávaný výraz bezchybný, při každém přečtení znaménka musí být v zásobníku připravena alespoň dvě čísla a po ukončení výpočtu musí v zásobníku zůstat číslo jediné — výsledná hodnota výrazu.

Algoritmus si nyní zapíšeme v Pascalu. Od vlastního vyhodnocování výrazu oddělíme technické detaily, jak vůbec získat zápis vyhodnocovaného výrazu a jak ho rozdělit na jednotlivé prvky. K tomu použijeme pomocnou funkci *Prvek*, která nám předá informaci, zda zpracovávaný výraz obsahuje ještě další prvek (prostřednictvím funkční hodnoty typu boolean). Pokud ano, funkce *Prvek* určí, zda jde o číslo nebo o operátor a prvek nám předá v příslušném výstupním parametru *Znak* nebo *Hodnota*. Pro úplnost si nejprve předvedeme, jak může vypadat taková funkce *Prvek*. Aby byla co nejjednodušší, budeme předpokládat, že zpracovávaný výraz je zadán na vstupu a že je v něm každá číselná konstanta následována mezerou.

```
function Prvek(var Hodnota:integer; var Znak:char): boolean;
  {pomocná funkce pro čtení výrazu ze vstupu po prvcích}
  {celý výraz musí být na vstupu zapsán na jednom řádku}
  {každé číslo musí být na vstupu následováno mezerou}
  {funkční hodnota - zda se ještě přečetl další prvek}
  {Hodnota, Znak - předání přečteného prvku}
  {pokud se četlo číslo, bude Znak = '$' (jako příznak)}
var Z: char;
    H: integer;
begin
if eof then
  Prvek := false
else
  begin
  Prvek := true;
  repeat
    read(Z)
  until Z <> ' ';
  if (Z >= '0') and (Z <= '9') then
    begin
    Znak := '$';   {na vstupu je číslo}
    H := 0;
    repeat
      H := H*10 + ord(Z) - ord('0');
      read(Z)
    until Z = ' ';
    Hodnota := H
    end
  else
    Znak := Z
```

```
  end
end;  {function Prvek}


function PostfixVyhodnoceni: integer;
  {vyhodnocení aritmet. výrazu zapsaného v postfixu}
  {využívá funkci Prvek pro vstup výrazu po částech}
const Max = 100;                {max. počet operandů ve výrazu}
var Zas: array[1..Max] of integer; {pracovní zásobník}
    V: 0..Max;                  {vrchol zásobníku Zas}
    H, H1, H2: integer;         {hodnoty operandů}
    Z: char;                    {znaménko na vstupu}
    Pokracovat: boolean;        {pokračování výpočtu}
begin
V := 0;
Pokracovat := Prvek(H,Z);
while Pokracovat do
  begin
  if Z = '$'then                {na vstupu je číslo H}
    begin
    V:=V+1; Zas[V]:=H;          {operand dáme na zásobník}
    Pokracovat := Prvek(H,Z)
    end
  else if V < 2 then            {chybný výraz!}
    begin
    Error;
    Pokracovat := false
    end
  else                          {na vstupu je znaménko Z}
    begin
    H2:=Zas[V]; V:=V-1;         {pravý operand ze zásobníku}
    H1:=Zas[V];                 {levý operand ze zásobníku}
    case Z of
      '+': H := H1 + H2;
      '-': H := H1 - H2;
      '*': H := H1 * H2;
      '/': H := H1 div H2;
    end;
    Zas[V] := H;                {výsledek dáme na zásobník}
    Pokracovat := Prvek(H,Z)
    end;
```



```
  end;
if V > 1 then
  Error
else
  PostfixVyhodnoceni := Zas[1]
end;  {function PostfixVyhodnoceni}
```

Rovněž k vyhodnocení výrazu v prefixové notaci použijeme lineární algoritmus a pomocný zásobník. Nabízejí se dva odlišné postupy vyhodnocování. Máme-li k dispozici najednou celý prefixový zápis výrazu a tento zápis není příliš dlouhý, můžeme si ho nejprve uložit do pole nebo do spojového seznamu a zápis pak zpracovávat po prvcích od konce směrem k začátku. Použijeme k tomu naprosto stejný postup jako při vyhodnocování výrazu v postfixové notaci. Jediná drobná změna se týká provádění nekomutativních operací — číslo vyzvednuté z vrcholu zásobníku bude nyní levým operandem, a ne pravým.

Druhá metoda vyhodnocování umožňuje číst prefixovou notaci výrazu po prvcích v normálním pořadí od začátku do konce. Od vyhodnocování postfixové notace se celý postup liší jednou malou komplikací: Do zásobníku se nyní budou ukládat nejen čísla, ale také operátory. Operátor přečtený ze vstupu vždy vložíme do zásobníku. Jestliže přečteme číselný operand, podíváme se, co je momentálně na vrcholu zásobníku. Pokud tam je znaménko, tento nový operand se ještě nemůže podílet na provedení operace, a uložíme ho proto do zásobníku. Je-li na vrcholu zásobníku číslo, vyjmeme ho ze zásobníku. Pod ním musí být nutně operátor, do zásobníku se nikdy nedostanou dvě čísla bezprostředně po sobě. Vyjmeme ze zásobníku i tento operátor o provedeme jemu příslušnou binární operaci s číslem z vrcholu zásobníku (levý operand) a s právě přečteným číslem (pravý operand). S výsledkem pak naložíme stejně jako s číslem přečteným ze vstupu, tj. podle momentálního vrcholu zásobníku ho buď uložíme do zásobníku, nebo se hned zúčastní další prováděné operace. Po zpracování celého zápisu výrazu zbyde v zásobníku jediné číslo, které udává výslednou hodnotu výrazu.

Předvedeme si, jak lze naprogramovat v Pascalu tuto druhou metodu. Opět budeme využívat funkci *Prvek* uvedenou výše.

```
function PrefixVyhodnoceni: integer;
  {vyhodnocení aritmet. výrazu zapsaného v prefixu}
  {využívá funkci Prvek pro vstup výrazu po částech}
const Max = 100;                    {max. počet prvků ve výrazu}
var Zas: array[1..Max] of           {pracovní zásobník}
          record H:integer; Z:char end;
    V: 0..Max;                      {vrchol zásobníku Zas}
    H, H1, H2: integer;             {hodnoty operandů}
    Z: char;                        {znaménko na vstupu}
    Pokracovat: boolean;            {pokračování výpočtu}
begin
V := 0;
Pokracovat := Prvek(H,Z);
while Pokracovat do
  begin
  if Z <> '$'then                   {na vstupu je operátor Z}
    begin
    V:=V+1; Zas[V].Z:=Z;            {operátor dáme na zásobník}
    Pokracovat := Prvek(H,Z)
    end
  else if V = 0 then                {na vstupu je číslo H}
    begin                           {zásobník je prázdný}
    V:=1;
    Zas[1].Z:='$'; Zas[1].H:=H; {číslo H dáme na zásobník}
    Pokracovat := Prvek(H,Z)
    end
  else if Zas[V].Z <> '$' then  {na vstupu je číslo H}
    begin                           {na vrcholu Zas je znaménko}
    V:=V+1;
    Zas[V].Z:='$'; Zas[V].H:=H; {číslo H dáme na zásobník}
    Pokracovat := Prvek(H,Z)
    end
  else if V < 2 then                {chybný výraz!}
    begin                           {na vstupu i na Zas je číslo}
    Error;
    Pokracovat := false
    end
  else                              {na vstupu i na Zas je číslo}
    begin
```



```
    H2 := H;                        {pravý operand ze vstupu}
    H1 := Zas[V].H; V:=V-1;         {levý operand ze zásobníku}
    Z := Zas[V].Z; V:=V-1;          {operátor ze zásobníku}
    case Z of
      '+': H := H1 + H2;
      '-': H := H1 - H2;
      '*': H := H1 * H2;
      '/': H := H1 div H2;
    end;    {výsledek necháme v H do dalšího průchodu cyklu}
    Z := '$'  {příště budeme mít "na vstupu" spočtené číslo}
    end;
  end;
if V > 1 then
  Error
else
  PrefixVyhodnoceni := Zas[1].H
end;  {function PrefixVyhodnoceni}
```

Vyhodnocení aritmetického výrazu zapsaného v běžné infixové notaci je obtížnější než oba předchozí případy. Existuje řada dosti rozdílných algoritmů, jak lze tuto úlohu řešit. Seznámíme se s myšlenkami alespoň některých z nich.

První možnou metodou vyhodnocování je zpracovávat výraz „zdola" a postupně v něm nahrazovat dílčí podvýrazy jejich hodnotami. Je to stejný postup, jaký jsme uváděli již v případě reprezentace výrazu binárním stromem. Přímo vyhodnotitelné podvýrazy je nyní ovšem třeba postupně vyhledávat v infixovém zápisu výrazu. Vyhledávání těchto podvýrazů je dosti složité a celý postup je zbytečně pomalý.

Také druhý algoritmus je obdobou toho, co jsme si již jednou předvedli při vyhodnocování výrazů uložených ve tvaru binárního stromu. Tentokrát budeme při vyhodnocování postupovat „shora" a budeme výraz rekurzivně rozkládat metodou „rozděl a panuj" (viz kap. 10). Ve výrazu nejprve vyhledáme operátor, který má být při vyhodnocování proveden až jako poslední. Tento operátor získáme poměrně snadno při jednom průchodu zápisem výrazu. Je to operátor nejnižší priority stojící zcela mimo závorky ve výrazu co nejvíce vpravo. Je to tentýž operátor, který je ve stromové reprezentaci výrazu umístěn v kořeni stromu. Celý vyhodnocovaný výraz se tímto operátorem rozdělí do dvou částí, a to na podvýraz stojící vlevo od vybraného operátoru a podvýraz vpravo od něj.

Každý z těchto podvýrazů samostatně vyhodnotíme (stejným postupem, pomocí rekurzivního volání) a s jejich hodnotami pak provedeme poslední operaci určenou vybraným operátorem. Tím získáme výslednou hodnotu celého výrazu.

Předchozí odstavec obsahuje v hrubých rysech celé řešení úlohy. Zbývá ještě dořešit dva dílčí problémy, které jsou však spíše technického rázu. Především posloupnost rekurzivních volání je také třeba někdy ukončit, aby byl výpočet konečný. K tomu dojde, pokud zkoumaný výraz již neobsahuje žádný operátor. Obsahuje tedy jen jediné číslo a to je přímo hodnotou výrazu. Při vyhledávání „posledního" operátoru ve výrazu by se také mohlo stát, že by zkoumaný výraz sice nějaké operátory obsahoval, ale že by žádný z nich nestál vně závorek. Tato situace nastane poměrně často při vyhodnocování výrazu s větším počtem závorek. V takovém případě je třeba nejprve odstranit nadbytečný pár závorek, který celý výraz uzavírá. Nutnost tohoto odstranění vnějšího páru závorek se může i vícekrát opakovat, neboť podle pravidel pro zápis výrazů je správný také výraz (((3 + 5))). Dejte však pozor na to, že pokud výraz začíná i končí závorkou, nemůžeme ještě automaticky tuto dvojici závorek odstranit. Takové dvě závorky totiž nemusí patřit k sobě, jak ukazuje třeba příklad výrazu $(5 + 9) * (10 - 7)$.

Postup vyhodnocování výrazu si můžeme ukázat na jednoduchém příkladu. Budeme vyčíslovat výraz $6 * (3 + 4)/2$. Mimo závorky stojí operátory „*" a „/", oba mají stejnou prioritu, takže vybereme „/" stojící ve výrazu více vpravo. Nyní budeme vyhodnocovat jeho levý podvýraz $6 * (3 + 4)$, podvýraz vpravo od „/" je již elementární a má hodnotu 2. Jediným operátorem mimo závorky v levém podvýrazu je „*". Vlevo od něj je elementární výraz s hodnotou 6. Vpravo od operátoru „*" stojí výraz $(3 + 4)$, který není elementární, ale neobsahuje žádný operátor mimo závorky. Sejmeme z něj proto vnější pár závorek, čímž dostaneme výraz $3 + 4$. Ten vyhodnotíme tak, že nalezneme jeho jediný operátor „+", oba jeho podvýrazy jsou již známé konstanty, takže můžeme ihned operátor „+" vykonat a spočítat hodnotu výrazu 7. Poté můžeme provést operaci „*", neboť již známe hodnoty jeho levého (6) i pravého (7) podvýrazu. Dostaneme hodnotu 42, která je hodnotou levého podvýrazu operátoru „/". Nakonec tedy provedeme závěrečné dělení a obdržíme výslednou hodnotu celého výrazu 21.

Tento druhý postup řešení je velmi elegantní, ale není nejrychlejší.



rekurze může jít v nepříznivém případě až do hloubky rovné počtu aritmetických operátorů ve výrazu a na každé úrovni rekurze se provádí vyhledávání, jehož pracnost je úměrná délce výrazu. Postup má proto v nejhorším případě kvadratickou časovou složitost.

Třetí algoritmus na vyhodnocování aritmetických výrazů vychází z toho, že umíme velmi snadno vykonávat jiné dvě činnosti. Dokážeme jednoduchým lineárním algoritmem převést infixový zápis výrazu do postfixové notace (viz závěr kap. 12.3) a stejně tak rychle v lineárním čase umíme vyhodnocovat postfixovou notaci. Spojením obou těchto postupů získáme algoritmus na vyhodnocování výrazů zapsaných v infixu, který má rovněž lineární časovou složitost. Není přitom ani třeba vytvářet nějakým způsobem ukládat celý postfixový zápis výrazu, oba dílčí algoritmy mohou pracovat souběžně. Toto řešení si ukážeme naprogramované v závěru kap. 12.3, až se seznámíme se zmíněným převáděním výrazu do postfixové notace.

Konečně čtvrtý postup je založen na analýze struktury výrazu. Strukturu aritmetického výrazu můžeme s ohledem na priority operátorů popsat následovně. Celý výraz je součtem (resp. rozdílem) několika členů, popř. je tvořen členem jediným. Každý člen je součinem (resp. podílem) několika faktorů, popř. je tvořen faktorem jediným. Každý faktor je buď přímo číselná konstanta, nebo je to výraz uzavřený do závorek.

Pro vyhodnocování výrazů si proto můžeme zavést soustavu funkcí, mezi nimiž je vztah nepřímé rekurze. Funkce *Vyraz* odpovídá za zpracování celého výrazu (popř. dílčího podvýrazu), výpočet provádí pomocí funkce *Clen*. Funkce *Clen* počítá hodnotu jednoho členu s využitím pomocné funkce *Faktor*. Funkce *Faktor* vyhodnocuje velmi jednoduše jeden faktor: buď je to přímo číselná konstanta, nebo narazí na závorku a pak nechá vyhodnotit dílčí podvýraz v závorce funkcí *Vyraz*. Algoritmus čte zpracovávaný výraz po prvcích zleva doprava a podle potřeby předává řízení mezi jednotlivými rekurzivními procedurami. Stejně jako předchozí řešení úlohy má i toto lineární časovou složitost (měřeno délkou výrazu).

Ukážeme si programovou realizaci tohoto algoritmu. Pro jednoduchost budeme předpokládat, že vyhodnocovaný aritmetický výraz je zadán korektně, a nebudeme kontrolovat jeho správnost. Pro vstup výrazu po jednotlivých prvcích budeme opět využívat pomocnou funkci *Prvek*, kterou jsme si popsali výše.

```
function Vyraz: integer;
  {funkce vyhodnocující aritmetický výraz zapsaný v infixu}
  {metoda - nepřímá rekurze funkcí Vyraz, Clen, Faktor}
  {využívá funkci Prvek pro vstup výrazu po částech}
  {předpokládá, že výraz na vstupu je syntakticky správný}
var V: integer;          {hodnota výrazu}
    H: integer;          {číslo na vstupu}
    Z: char;             {znaménko na vstupu}
    Pokracovat: boolean;
    NactenoZnamenko: boolean;      {pro signál z funkce Clen}

 function Faktor: integer;
   {pomocná funkce na vyhodnocení jednoho faktoru}
   {faktorem je číselná hodnota nebo výraz v závorkách}
 var H: integer;         {číslo na vstupu}
     Z: char;            {znaménko na vstupu}
 begin
 if Prvek(H,Z) then
   if Z = '$' then       {faktorem je přímo číselná konstanta}
     Faktor := H
   else                  {Z='(', faktorem je výraz v závorkách}
     Faktor := Vyraz
 else
   Error                 {chybný výraz!}
 end;  {function Faktor}

function Clen: integer;
  {pomocná funkce na vyhodnocení jednoho členu}
  {členem je jeden faktor nebo součin/podíl více faktorů}
  {někdy nechává v Z načtené další znaménko + nebo -}
  { {...} o tom signál v globálním NactenoZnamenko}
var H: integer;              {číslo na vstupu - nevyužívá se}
    C: integer;                                {hodnota členu}
    Pokracovat: boolean;
begin
NactenoZnamenko := false;
C := Faktor;
Pokracovat := Prvek(H,Z);
while Pokracovat do
  if Z = '*' then                {součin faktorů}
```



```
    begin
    C := C * Faktor;
    Pokracovat := Prvek(H,Z)
    end
  else if Z = '/' then           {podíl faktorů}
    begin
    C := C div Faktor;
    Pokracovat := Prvek(H,Z)
    end
  else                           {nutně Z='+', '-' nebo ')' }
    begin
    Pokracovat := false;         {konec členu}
    NactenoZnamenko := true      {signál pro Vyraz}
    end;
 Clen := C
end;  {function Clen}

begin  {function Vyraz}
V := Clen;
if NactenoZnamenko then
  Pokracovat := true
else
  Pokracovat := Prvek(H,Z);
while Pokracovat do
  if Z = '+' then                {součet členů}
    begin
    V := V + Clen;
    if not NactenoZnamenko then
      Pokracovat := Prvek(H,Z)
    end
  else if Z = '-' then           {rozdíl členů}
    begin
    V := V - Clen;
    if not NactenoZnamenko then
      Pokracovat := Prvek(H,Z)
    end
  else                           {nutně Z=')'}
    Pokracovat := false;         {konec výrazu}
Vyraz := V
end;  {function Vyraz}
```

## 12.3 Převody mezi aritmetickými notacemi

V kap. 12.1 jsme poznali několik různých reprezentací aritmetických výrazů a v kap. 12.2 jsme si ukázali, jak se tyto výrazy vyhodnocují. Pro úplnost nám ještě chybí naučit se jednotlivé podoby aritmetického výrazu mezi sebou převádět.

Procedury *Prefix*, *Postfix* a *Infix* uvedené v závěru kap. 12.1 zajišťují převod aritmetického výrazu z vnitřní reprezentace pomocí binárního stromu do textového zápisu v jednotlivých notacích. Nyní se naučíme vybudovat binární strom z prefixové, postfixové a infixové notace výrazu. Tím budeme zároveň umět jednotlivé notace mezi sebou převádět. Převod se uskuteční jednoduše tak, že nejprve z výchozí notace vytvoříme binární strom a pak při patřičném průchodu tímto stromem získáme zápis ve výsledné notaci.

Algoritmy pro vytvoření binárního stromu z textového zápisu aritmetického výrazu jsou velmi podobné algoritmům pro vyhodnocování výrazu, které jsme uvedli v kap. 12.2. Způsob zpracování zápisu výrazu se vůbec neliší od vyhodnocování, pouze místo vlastního provádění jednotlivých operací s již spočtenými operandy budeme vytvářet odspodu strom reprezentující výraz. Kdykoli se při vyhodnocování výrazu objevila číselná konstanta, vytvoříme nyní nový uzel stromu obsahující tuto konstantu a zaznamenáme si odkaz na ni. Když se během vyhodnocování prováděla některá aritmetická operace obsažená ve výrazu, vytvoříme teď nový uzel stromu obsahující znak této operace a jako následníky mu připojíme odkazy na příslušné operandy. Tuto úpravu si ukážeme již jen na nejjednodušším případu, tj. na postfixové notaci. Úprava algoritmů z kap. 12.2 pro prefixovou a infixovou notaci je obdobná.

Zápis výrazu v postfixové notaci budeme opět číst po prvcích zleva doprava. Při práci budeme používat pomocný zásobník ukazatelů na uzly stromu. Jestliže přečteme prvek tvořený číselným operandem, vygenerujeme nový uzel (budoucí list stromu), do obou jeho ukazatelů vložíme hodnotu nil, přiřadíme mu právě přečtenou číselnou hodnotu a odkaz na něj vložíme do zásobníku. Kdykoli je nový prvek operátorem, vygenerujeme pro něj nový vnitřní uzel stromu se znaménkem, ze zásobníku vyzvedneme vrchní dvě položky (ukazatele na uzly stromu) a přiřadíme je do ukazatelů právě vytvářeného uzlu. Ukazatel z vrcholu zásobníku se přiřadí do pravého následníka nového uzlu a druhý ukazatel ze zásobníku do levého. Potom vložíme do zásobníku ukazatel na nově



tvořený uzel. Po zpracování celého výrazu zůstane v zásobníku jediná ložka — ukazatel na kořen celého stromu reprezentujícího výraz.

Algoritmus zapíšeme v Pascalu, opět s využitím pomocné funkce *ek*.

```
ction PostfixStrom: Uk;
{vytvoření binárního stromu pro aritmetický výraz}
{  zapsaný v postfixové notaci}
{využívá funkci Prvek pro vstup výrazu po částech}
st Max = 100;                    {max. počet operandů ve výrazu}
r Zas: array[1..Max] of Uk;      {pracovní zásobník}
  V: 0..Max;                     {vrchol zásobníku Zas}
  H: integer;                    {hodnota operandu na vstupu}
  Z: char;                       {znaménko na vstupu}
  Pokracovat: boolean;           {pokračování výpočtu}
  U: Uk;
gin
 := 0;
Pokracovat := Prvek(H,Z);
hile Pokracovat do
  begin
  new(U);                        {nový uzel pro nový prvek}
  if Z = '$'then                 {na vstupu je číslo H}
    begin
    U^.Hodnota := H;
    U^.L:=nil; U^.P:=nil;        {bude to list}
    V:=V+1; Zas[V]:=U;           {operand dáme na zásobník}
    Pokracovat := Prvek(H,Z)
    end
  else if V < 2 then             {chybný výraz!}
    begin
    Error;
    Pokracovat := false
    end
  else                           {na vstupu je znaménko Z}
    begin
    U^.Znak := Z;
    U^.P := Zas[V]; V:=V-1;      {pravý operand ze zásobníku}
    U^.L := Zas[V];              {levý operand ze zásobníku}
    Zas[V] := U;                 {výsledek dáme na zásobník}
    Pokracovat := Prvek(H,Z)
```

```
    end;
  end;
if V > 1 then
  Error
else
  PostfixStrom := Zas[1]
end;  {function PostfixStrom}
```

Ze všech převodů mezi notacemi aritmetického výrazu je nejdůležitější převádění výrazu z infixové podoby do postfixové notace. V běžném infixovém tvaru je totiž zpracovávaný výraz obvykle zadán od uživatele, zatímco postfixový zápis je nejvýhodnější pro vyhodnocování či pro jiné další zpracování výrazu. Ukážeme si proto ještě jiný postup, který převádí **infixový zápis na postfixový přímo**, bez mezikroku přes vnitřní reprezentaci výrazu binárním stromem. Tento algoritmus je navíc velmi rychlý, má lineární časovou složitost (měřeno délkou zpracovávaného výrazu).

Algoritmus čte infixový zápis výrazu po prvcích zleva doprava. Jedním prvkem tentokrát rozumíme nejen číselnou konstantu nebo binární operátor, ale také levou či pravou závorku. K výpočtu se využívá pracovní zásobník na odkládání znamének, tj. operátorů a závorek. V postfixovém zápisu výrazu mají všechny operandy stejné vzájemné pořadí jako v zápisu infixovém. Operátory se ovšem oproti infixu musí pozdržet, aby následovaly až za „svými" operandy. Závorky určují způsob pozdržování jednotlivých operátorů, do postfixového zápisu se ale již neukládají. Při převádění tedy budeme postupovat následovně. Kdykoli přečteme z infixového zápisu číselnou konstantu, přímo ji zapíšeme do vytvářeného postfixového zápisu. Jestliže přečteme aritmetický operátor, vložíme ho do zásobníku. Předtím ale ještě ze zásobníku vyjmeme a zapíšeme na výstup všechny operátory stejné nebo vyšší priority, které tam na vrchu jsou. Výraz je totiž vyhodnocován podle priorit a operátory stejné priority zleva doprava, takže žádný operátor nesmí v postfixovém zápisu „předběhnout" jiný operátor stejné nebo vyšší priority. Pokud přečteme ze vstupu levou závorku, uložíme ji vždy do zásobníku. Až později přečteme závorku pravou, vyjmeme postupně ze zásobníku a zapíšeme na výstup všechny operátory ležící nad zaznamenanou levou závorkou. Tuto závorku ze zásobníku také vyzvedneme, ale do výstupního postfixového zápisu ji nevkládáme.



Celý postup zapíšeme ve tvaru procedury v jazyce Pascal. Pro vstup jednotlivých prvků tvořících infixový zápis výrazu použijeme opět naši pomocnou funkci *Prvek*. Výsledný zápis výrazu v postfixové notaci budeme přímo vypisovat. Procedura slouží pouze pro převod mezi notacemi, ne pro kontrolu správnosti vstupního infixového zadání výrazu. Pro jednoduchost nekontroluje správnost vstupního výrazu, sleduje pouze správnou strukturu uzávorkování.

```
procedure InfixNaPostfix;
  {převod aritmet. výrazu zapsaného v infixu do postfixu}
  {využívá funkci Prvek pro vstup výrazu po částech}
  {výsledný postfixový zápis výrazu přímo vypisuje}
  {nekontroluje správnost vstupního infixového zápisu}
const Max = 100;              {max. počet operátorů ve výrazu}
var Znam: array[0..Max] of char;{pracovní zásobník}
    W: 0..Max;                {vrchol zásobníku Znam}
    H: integer;               {hodnoty operandů}
    Z: char;                  {znaménko na vstupu}
    Pokracovat: boolean;      {pokračování výpočtu}
begin
W := 0;
Pokracovat := Prvek(H,Z);
while Pokracovat do
  begin
  case Z of
  '$': write(H, ' ');         {na vstupu je číslo H}
  '(': begin
       W:=W+1; Znam[W]:='('   {závorku do zásobníku}
       end;
  ')': begin                  {ze zásobníku až k závorce}
       while (W > 0) and (Znam[W] <> '(') do
         begin
         write(Znam[W]); W:=W-1
         end;
       if W = 0 then
         Error                {chybný výraz!}
       else
         W:=W-1               {ještě zrušit závorku}
       end;
  '*', '/':          {operátory stejné priority ven ze Znam}
```

```
      begin
      while (W > 0) and (Znam[W] in ['*','/']) do
        begin
        write(Znam[W]); W:=W-1
        end;
      W:=W+1; Znam[W]:=Z        {uložit Z na zásobník}
      end;
  '+', '-': {operátory stejné a vyšší priority ven ze Znam}
      begin
      while (W > 0) and (Znam[W] in ['+','-','*','/']) do
        begin
        write(Znam[W]); W:=W-1
        end;
      W:=W+1; Znam[W]:=Z        {uložit Z na zásobník}
      end;
  end; {case}
  Pokracovat := Prvek(H,Z)
  end;
while (W > 0) and (Znam[W] <> '(') do
  begin
  write(Znam[W]); W:=W-1        {ještě vyprázdnit zásobník}
  end;
writeln;
if W > 0 then
  Error                         {chybný výraz!}
end;  {procedure InfixNaPostfix}
```

Uvedený algoritmus na převádění infixové notace aritmetického výrazu do postfixové můžeme přímo spojit s algoritmem vyhodnocování výrazu zapsaného v postfixové notaci, který jsme si předvedli v kap. 12.2. Získáme tak rychlý a elegantní postup pro vyhodnocení výrazu zapsaného v běžném infixovém tvaru. Algoritmus má lineární časovou složitost stejně jako obě komponenty, jejichž složením vznikl. Při své práci používá dva zásobníky — jeden na ukládání operátorů (převod výrazu do postfixu), druhý pro ukládání čísel (během vlastního vyhodnocování). Vytvářená postfixová forma výrazu se vůbec neukládá jako celek, nýbrž je průběžně ihned vyhodnocována.



```
function InfixVyhodnoceni: integer;
{vyhodnocení aritmet. výrazu zapsaného v infixu}
{metoda založená na převodu přes postfix}
{využívá funkci Prvek pro vstup výrazu po částech}
{nekontroluje správnost vstupního infixového zápisu}
const Max = 100;                {max. počet prvků ve výrazu}
var Znam: array[0..Max] of char;{zásobník na znaménka}
  W: 0..Max;                    {vrchol zásobníku Znam}
  Zas: array[1..Max] of integer; {zásobník na čísla}
  V: 0..Max;                    {vrchol zásobníku Zas}
  H: integer;                   {operand na vstupu}
  Z: char;                      {znaménko na vstupu}
  Pokracovat: boolean;          {pokračování výpočtu}

procedure Proved(Z: char);
{vykonání operace spojené s operátorem Z}
var H, H1, H2: integer;         {hodnoty operandů}
begin
if V < 2 then                   {chybný výraz!}
  Error
else
  begin
  H2:=Zas[V]; V:=V-1;           {pravý operand ze zásobníku}
  H1:=Zas[V];                   {levý operand ze zásobníku}
  case Z of
    '+': H := H1 + H2;
    '-': H := H1 - H2;
    '*': H := H1 * H2;
    '/': H := H1 div H2;
  end;
  Zas[V] := H;                  {výsledek dáme na zásobník}
  end;
end;  {procedure Proved}

begin
W := 0;
V := 0;
Pokracovat := Prvek(H,Z);
while Pokracovat do
```

```
  begin
  case Z of
  '$': begin                          {na vstupu je číslo H}
       V:=V+1; Zas[V]:=H;             {operand dáme na zásobník}
       end;
  '(': begin                          {závorku do zásobníku}
       W:=W+1; Znam[W]:='('
       end;
  ')': begin                          {ze zásobníku až k závorce}
       while (W > 0) and (Znam[W] <> '(') do
         begin
         Proved(Znam[W]); W:=W-1
         end;
       if W = 0 then
         Error                        {chybný výraz!}
       else
         W:=W-1                       {zrušit závorku}
       end;
  '*', '/':          {operátory stejné priority ven ze Znam}
       begin
       while (W > 0) and (Znam[W] in ['*','/']) do
         begin
         Proved(Znam[W]); W:=W-1
         end;
       W:=W+1; Znam[W]:=Z             {uložit Z na zásobník}
       end;
  '+', '-':  {operátory stejné a vyšší priority ven ze Znam}
       begin
       while (W > 0) and (Znam[W] in ['+','-','*','/']) do
         begin
         Proved(Znam[W]); W:=W-1
         end;
       W:=W+1; Znam[W]:=Z             {uložit Z na zásobník}
       end;
  end; {case}
  Pokracovat := Prvek(H,Z)
  end;
while (W > 0) and (Znam[W] <> '(') do
  begin
  Proved(Znam[W]); W:=W-1             {ještě vyprázdnit zásobník}
  end;
```



```
writeln;
if W > 0 then
  Error;                              {chybný výraz!}
if V > 1 then
  Error
else
  InfixVyhodnoceni := Zas[1]
end; {function InfixVyhodnoceni}
```

## CVIČENÍ

**1.** Upravte všechny programy uvedené v kap. 12 tak, aby na místě číselných konstant ve výrazech bylo možné zadávat desetinná čísla a aby všechny výpočty probíhaly v proměnných typu real.

**2.** Upravte všechny programy uvedené v kap. 12 tak, aby ve výrazech bylo možné používat i unární znak minus. *Návod*: Při zápisu aritmetického výrazu v prefixové nebo postfixové notaci je nutné textově odlišit unární minus od binárního. Jinak by nebylo možné výrazy vyhodnocovat. Místo unárního minusu používejte v prefixovém a postfixovém zápisu jiný znak než ‚–', například ‚@'.

**3.** Naprogramujte algoritmus vyhodnocení aritmetického výrazu zapsaného v běžné infixové notaci metodou „rozděl a panuj". Podrobný popis postupu najdete v kap. 12.2.

# 13 EFEKTIVITA REKURZIVNÍCH ALGORITMŮ

Rekurze je velmi silný a užitečný nástroj programátora, je to však také nástroj nebezpečný a při jeho použití musíme být obzvlášť opatrní. Mechanické použití rekurze při řešení úlohy vede často k algoritmům s exponenciální časovou složitostí. Programy realizující takové algoritmy jsou pak velice pomalé a jsou prakticky použitelné pouze pro řešení „malých" úloh. V programech využívajících rekurzivního volání také velmi snadno uděláte chybu a při ladění programu se takováto chyba obvykle hůře hledá a odstraňuje než u programů bez rekurze.

Rekurzi bychom neměli používat, pokud již samotný charakter úlohy nevede přirozeným způsobem k rekurzivnímu řešení. Nesprávné a nevhodné je její použití také tehdy, jestliže úlohu dokážeme stejně dobře řešit i bez rekurze. V takovém případě je rekurze zbytečná a zpravidla vede jenom ke zvýšení časových a paměťových nároků programu při výpočtu. Naproti tomu jsme se již i v této knize setkali s úlohami, které se pomocí rekurze řešily naprosto přirozeně a u nichž bylo použití rekurze zcela na místě. Jako příklad můžeme uvést techniku prohledávání do hloubky (kap. 8) včetně její aplikace při řešení grafových úloh (kap. 9), na rekurzi je založena metoda „rozděl a panuj" (kap. 10), která se využívá například v některých třídicích algoritmech (kap. 11).

Jestliže se při řešení nějaké úlohy rozhodneme pro použití rekurze, musíme věnovat velkou pozornost efektivitě výsledného algoritmu. V některých případech nemůžeme exponenciální časové složitosti nijak zabránit, neboť je dána přímo charakterem úlohy. Tak je tomu například při průchodu stavovým prostorem do hloubky (backtracking — kap. 8.2). I v takovém případě se ale snažíme alespoň výpočet co nejvíce urychlit (ořezávání, heuristiky — kap. 8.3). Často je však možné efektivitu rekurzivního algoritmu velmi zásadně vylepšit. Stává se totiž, že při nešikovném použití rekurze v řešení úlohy dochází k opakování téhož výpočtu zbytečně několikrát. Pokud rekurzivní procedura nebo funkce počítá nějakou hodnotu a při výpočtu potřebuje vícekrát použít stejnou pomocnou hodnotu, která je výsledkem rekurzivního volání, je zbytečné zjišťovat tuto pomocnou hodnotu vždy znovu pomocí nového rekurzivního volání. Celý výpočet se značně urychlí, jestliže si všechny již jednou spočítané hodnoty budeme ukládat do pomocného pole.



## 13.1 Fibonacciho čísla

V předchozím odstavci jsme několika větami popsali základní princip zvyšování efektivity rekurzivních algoritmů. Tuto metodu si nyní ukážeme na jedné úloze dobře známé ze všech učebnic programování. Je to úloha nalézt $N$-té Fibonacciho číslo pro dané $N$, $N \geq 0$. Nejprve si ale musíme Fibonacciho čísla přesně definovat. Označíme-li $Fib(N)$ v pořadí $N$-té Fibonacciho číslo, pak platí:

$$Fib(0) = 0$$
$$Fib(1) = 1$$
$$Fib(N) = Fib(N-1) + Fib(N-2) \quad \text{pro } N > 1.$$

Nekonečná posloupnost Fibonacciho čísel tedy začíná čísly

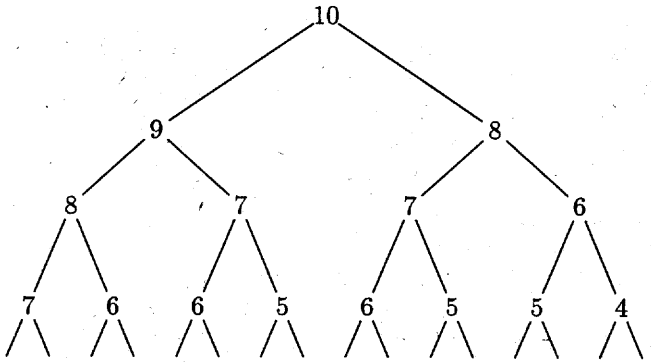
0 1 1 2 3 5 8 13 21 34 55 89 ...

Matematická definice Fibonacciho čísel je rekurzivní, takže na první pohled se zdá přirozené použít k výpočtu $N$-tého Fibonacciho čísla rekurzivní funkci. Získáme ji velmi snadno přímým přepsáním uvedené definice do programovacího jazyka.

```
function Fib1(N: integer): integer;
  {výpočet N-tého Fibonacciho čísla prostou rekurzí}
  begin
  if (N=0) or (N=1) then
    Fib1 := N
  else
    Fib1 := Fib1(N-1) + Fib1(N-2)
  end;  {function Fib1}
```

Pokud se zamyslíte nad tímto řešením, rychle zjistíte, že má exponenciální časovou složitost a že se v něm provádí velké množství zbytečných výpočtů. Například pro určení desátého Fibonacciho čísla musíme nejprve zjistit deváté a osmé, přičemž k výpočtu devátého si musíme nejprve spočítat osmé a sedmé a k určení osmého sedmé a šesté

atd. Během výpočtu *Fib1*(10) je tedy dvakrát voláno *Fib1*(8), třikrát *Fib1*(7), pětkrát *Fib1*(6) atd.




*Obr. 29* Strom rekurzivních volání funkce *Fib1* pro výpočet desátého Fibonacciho čísla

Pro zásadní zrychlení výpočtu postačí, jestliže tentýž algoritmus doplníme o jedno pomocné pole $F$. Do něj si budeme průběžně ukládat všechna již jednou spočítaná Fibonacciho čísla. Před každým rekurzivním voláním se podíváme, zda by takové volání nebylo zbytečné, zda hledanou hodnotu již nemáme v poli $F$. Výsledný algoritmus má rázem lineární časovou složitost. Hledané v pořadí $N$-té Fibonacciho číslo totiž závisí pouze na Fibonacciho číslech menších a každé z nich počítáme nejvýše jednou. Tato úprava řešení má jen jedinou drobnou nevýhodu. Abychom mohli zavést pole $F$, musíme předem pevně zvolit jeho velikost a tím stanovíme omezení, pro jak velká $N$ je naše funkce použitelná. V tomto konkrétním případě však toto omezení nehraje žádnou roli. Posloupnost Fibonacciho čísel roste tak rychle, že hranice použitelnosti funkce je dána spíše hodnotou největšího zobrazitelného celého čísla (v Pascalu hodnotou konstanty `maxint`).

Řešení si ukážeme zapsané v Pascalu, a to ve dvou variantách. Funkce *Fib2* má kratší zápis a je bližší původní funkci *Fib1*, funkce *Fib3* provádí při výpočtu o něco méně rekurzivních volání.



```
const MaxN = 100;                    {maximální možná hodnota N}
var F: array[0..MaxN] of integer; {již spočítaná Fib. čísla}

{inicializace pole F:}
F[0] := 0;
F[1] := 1;
for I:=2 to N do F[I] := -1;          {hodnoty zatím neznáme}

function Fib2(N: integer): integer;
  {výpočet N-tého Fibonacciho čísla rekurzí s pomocným
   polem pro uložení již známých hodnot}
  begin
  if F[N] = -1 then                  {zatím N-té číslo neznáme}
    F[N] := Fib2(N-1) + Fib2(N-2);
  Fib2 := F[N]
  end;  {function Fib2}

function Fib3(N: integer): integer;
  {výpočet N-tého Fibonacciho čísla rekurzí s pomocným
   polem pro uložení již známých hodnot}
  begin
  if (N=0) or (N=1) then
    Fib3 := N
  else
    begin
    if F[N-1] = -1 then F[N-1] := Fib3(N-1);
    if F[N-2] = -1 then F[N-2] := Fib3(N-2);
    Fib3 := F[N-1] + F[N-2]
    end
  end;  {function Fib3}
```

Úlohu nalézt $N$-té Fibonacciho číslo je možné řešit také bez použití rekurze, postupným výpočtem Fibonacciho čísel v cyklu od nejmenšího až k $N$-tému. To je jistě nejlepší řešení úlohy. Má lineární časovou složitost, nepotřebuje žádné pomocné pole a neztrácí ani čas prováděním rekurzivních volání. Algoritmus nejsnáze zapíšeme pomocí celočíselných proměnných $X$, $Y$, $Z$, které slouží k uložení tří posledních Fibonacciho čísel. Další pomocná proměnná $I$ se v cyklu zvětšuje od 1 do $N$ a určuje, kolikáté Fibonacciho číslo právě počítáme. V pořadí $I$-té číslo vždy

spočítáme jako součet čísel s pořadím $I-1$ (je uloženo v proměnné $Y$) a $I-2$ (to je uschováno v proměnné $Z$) a uložíme ho do proměnné $X$. Při zvýšení hodnoty proměnné $I$ o 1 se samozřejmě musí „posunout" také uložená Fibonacciho čísla. Starou hodnotu proměnné $Z$ již nebudeme potřebovat, do $Z$ vložíme číslo $Y$ a pak do $Y$ číslo z proměnné $X$. Tím je proměnná $X$ uvolněna a připravena pro výpočet dalšího Fibonacciho čísla. Celý postup zapsaný v Pascalu ukazuje funkce *Fib4*.

```
function Fib4(N: integer): integer;
  {výpočet N-tého Fibonacciho čísla cyklem}
  var I, X, Y, Z: integer;
  begin
  if N=0 then
    Fib4 := 0
  else
    begin
    I := 1;
    X := 1; Y := 0;
    while I < N do
      begin
      I := I + 1;        {kolikáté Fib. číslo počítáme}
      Z := Y; Y := X;    {posun uschovaných hodnot předchozích
                          dvou Fibonacciho čísel}
      X := Y + Z         {určení I-tého Fibonacciho čísla}
      end;
    Fib4 := X
    end
  end;  {function Fib4}
```

Na závěr kapitoly o výpočtu Fibonacciho čísel bude dobré učinit ještě několik technických poznámek. Pokud bychom chtěli vyšší Fibonacciho čísla skutečně počítat na počítači s využitím výše uvedených algoritmů, nesmíme zapomenout, že posloupnost Fibonacciho čísel velmi rychle roste. Při zobrazení celých čísel se znaménkem do 16 bitů, které se používá například pro uložení proměnných typu `integer` v Turbo Pascalu na počítačích PC, můžeme naše ukázkové programy použít k určení nejvýše 23. Fibonacciho čísla (je rovno 28 657). Následující 24. Fibonacciho číslo již nelze spočítat, neboť přesahuje hodnotu `maxint`. Museli bychom proto použít jiný způsob uložení celých čísel a výpočtů s nimi. V Turbo Pascalu



existuje typ `longint` pro uložení celých čísel se znaménkem do 32 bitů (ten nám ale postačí nejvýše na 46. Fibonacciho číslo), nebo si můžeme naprogramovat vlastní „vícenásobnou aritmetiku".

Je také zajímavé otestovat přímo na počítači jednotlivé uvedené algoritmy a porovnat je z hlediska rychlosti výpočtu. Funkce *Fib2*, *Fib3* a *Fib4* počítají i velmi vysoká Fibonacciho čísla ve zlomcích sekundy. Naproti tomu exponenciální časová složitost funkce *Fib1* se začíná viditelně projevovat poměrně brzo, na běžném počítači typu PC zhruba kolem dvacátého Fibonacciho čísla. Hranice praktické použitelnosti funkce *Fib1* leží přibližně u třicátého Fibonacciho čísla, jehož výpočet trvá řádově několik minut (samozřejmě podle typu počítače).

## 13.2 Silniční síť

Stejně jako při hledání Fibonacciho čísel (kap. 13.1) budeme postupovat i při řešení následující úlohy o silniční síti. Na příkladech o síti silnic mezi městy jsme již dříve demonstrovali řadu jiných algoritmů, tentokrát však půjde o silniční síť dosti zvláštního tvaru. Tato úloha pochází z kategorie programování matematické olympiády a její rozbor lze nalézt také v knize [14].

Mezi $N$ městy označenými čísly 1, 2, ..., $N$ je vybudována silniční síť těchto vlastností:

- všechny silnice jsou jednosměrné
- z každého města vycházejí nejvýše dvě silnice
- jestliže z nějakého města vycházejí dvě silnice, tyto silnice vedou do různých měst
- každá silnice vede vždy z města s nižším číslem do města s vyšším číslem

Důsledkem poslední uvedené vlastnosti je, že do města číslo 1 nevede žádná silnice (nemá odkud) a z města $N$ nevede žádná silnice (nemá kam). Naším úkolem je určit, kolik existuje různých cest z města 1 do města $N$. Dvě cesty z města 1 do města $N$ považujeme za různé, jestliže se liší alespoň v některém svém úseku.

Budeme předpokládat, že silniční síť je zadána pomocí dvou polí celých čísel $X[1..N]$ a $Y[1..N]$. Hodnoty $X[M]$ a $Y[M]$ udávají, kam směřují silnice vedoucí z města $M$. Pokud z města $M$ vede jen jedna silnice, je číslo jejího cílového města uloženo v $X[M]$, zatímco hodnota

$Y[M]$ je nulová. Jestliže z nějakého města $M$ nevede žádná silnice, jsou nulové obě hodnoty $X[M]$, $Y[M]$. Z vlastností silniční sítě plyne, že pro všechna $M$, $1 \leqq M \leqq N$, platí podmínky:

- $M < X[M] \leqq N$ nebo $X[M] = 0$
- $M < Y[M] \leqq N$ nebo $Y[M] = 0$
- $X[M] \neq Y[M]$ nebo $X[M] = Y[M] = 0$.

K vyjádření hledaného počtu různých cest z města 1 do města $N$ snadno odvodíme rekurentní vztah a na jeho základě pak napíšeme výslednou rekurzivní funkci. Při určování počtu cest můžeme postupovat dvěma způsoby, buď od města 1 k městu $N$, nebo naopak nazpět od města $N$ k městu 1. Nejprve navrhneme funkci *Cesty1*, která bude pro zadané $K$ určovat počet různých cest vedoucích z města 1 do města $K$. Údaj $K$ bude parametrem této funkce. Pro $K = 1$ bude výsledkem jistě 1, neboť z města 1 do města 1 se dostaneme jediným způsobem — bez cestování. Jestliže $K > 1$, musíme nejprve nalézt všechny silnice končící ve městě $K$. Zjistíme, odkud vedou, a potom již určíme výsledek jako součet počtů různých cest vedoucích z města 1 do těchto měst. Tyto dílčí počty určíme pomocí rekurzivního volání funkce *Cesty1*.

Správnost uvedeného řešení je zajištěna podmínkou, že každá silnice vede vždy z města s nižším číslem do města s vyšším číslem. Díky této podmínce se posloupnost rekurzivních volání nezacyklí, funkce *Cesty1* je volána se stále menšími hodnotami parametru. Pro vyřešení celé úlohy bude funkce *Cesty1* zavolána s parametrem rovným $N$. Funkci *Cesty1* nyní zapíšeme v Pascalu.

```
function Cesty1(K: integer): integer;
  {počet různých cest z města 1 do města K - prostá rekurze}
  var Pocet, I: integer;
  begin
  if K = 1 then
    Cesty1 := 1                {jsme na místě - jediná možnost}
  else
    begin
    Pocet := 0;                          {počítadlo možných cest}
    for I:=1 to K-1 do
      if (X[I]=K) or (Y[I]=K) then   {vede silnice z I do K}
        Pocet := Pocet + Cesty1(I);
    Cesty1 := Pocet
```



```
    end
  end;  {function Cesty1}
```

Opačný postup řešení představuje rekurzivní funkce *Cesty2*, která pro zadané $K$ počítá, kolik existuje různých cest z města $K$ do města $N$. Celou úlohu vyřešíme voláním funkce *Cesty2* s parametrem 1. Funkce *Cesty2* dává pro $K = N$ výsledek 1, neboť z města $N$ do města $N$ se dostaneme jediným způsobem. Pokud $K < N$, určíme nejprve města, do nichž vede přímá silnice z města $K$. Výsledkem pak bude součet počtů různých cest vedoucích z těchto měst do města $N$. Funkce *Cesty2* je navržena zcela analogicky jako *Cesty1*. Vzhledem ke zvolenému způsobu uložení informací o silniční síti je však funkce *Cesty2* o něco vhodnější a rychlejší. Čísla měst přímo dostupných z města $K$ totiž získáme jednoduše jako $X[K]$, $Y[K]$, zatímco čísla měst, z nichž je číslo $K$ přímo dostupné, jsme museli vyhledávat při průchodu celými poli $X$ a $Y$.

```
function Cesty2(K: integer): integer;
  {počet různých cest z města K do města N - prostá rekurze}
  var Pocet: integer;
  begin
  if K = N then
    Cesty2 := 1                {jsme na místě - jediná možnost}
  else
    begin
    Pocet := 0;                          {počítadlo možných cest}
    if X[K] > 0 then Pocet := Pocet + Cesty2(X[K]);
    if Y[K] > 0 then Pocet := Pocet + Cesty2(Y[K]);
    Cesty2 := Pocet
    end
  end;  {function Cesty2}
```

Oba dva rekurzivní postupy, které jsme si ukázali, mají pro některé tvary silniční sítě exponenciální časovou složitost. Výsledný počet různých cest z města 1 do města $N$ může být exponenciálně závislý na počtu měst $N$. Přitom výsledek získáme jak v případě funkce *Cesty1*, tak i u funkce *Cesty2*, postupným nasčítáním ze samých jedniček. Ty vznikají jakožto funkční hodnoty při voláních *Cesty1*(1), resp. *Cesty2*($N$). Celkový počet provedených volání rekurzivní funkce potřebný ke stanovení výsledku tudíž může růst exponenciálně v závislosti na $N$. Důvody

této vysoké časové složitosti jsou naprosto stejné jako při rekurzivním výpočtu Fibonacciho čísel v kap. 13.1. Funkce *Cesty1* (resp. *Cesty2*) je během výpočtu mnohokrát opakovaně volána se stejnou hodnotou svého vstupního parametru. V důsledku toho se některé hodnoty *Cesty1*$(i)$, resp. *Cesty2*$(i)$ počítají zbytečně vícekrát. Přitom jejich výpočet představuje často velké množství práce, neboť v sobě může obsahovat další rekurzivní volání.

Stejně jako v kap. 13.1 můžeme výpočet velmi urychlit, jestliže použijeme pomocné pole $C[1..N]$ k ukládání již spočtených hodnot rekurzivní funkce. Tuto úpravu si ukážeme pouze pro případ funkce *Cesty2*, která je pro naši úlohu výhodnější než funkce *Cesty1*. Získáme tak novou rekurzivní funkci *Cesty3*. Funkce *Cesty3* bude tedy také počítat pro dané $K$ počet různých cest vedoucích z města $K$ do města $N$. Hodnoty pomocného pole $C$ budou mít stejný význam, číslo $C[K]$ je rovno počtu různých cest vedoucích z města $K$ do města $N$. Funkce *Cesty3* se liší od *Cesty2* pouze tím, že před každým provedením rekurzivního volání ověří pomocí pole $C$, zda již příslušná hodnota nebyla někdy dříve spočítána. Jestliže ano, rekurzivní volání je nahrazeno přímým vyzvednutím hodnoty z pole $C$.

Touto úpravou předchozího řešení se nic nezmění na správnosti výsledku. Výpočet probíhá stejným způsobem, jenom se snížil počet provedených rekurzivních volání. Funkce *Cesty3* bude rekurzivně zavolána pouze k určení dosud neznámé hodnoty, tedy nejvýše $N$-krát. Celé řešení má proto lineární časovou složitost.

```
var C: array[1..N] of integer;   {počet cest vedoucích do N}

{inicializace pole C:}
for I:= 1 to N-1 do C[I] := -1;   {hodnota dosud není známa}
C[N] := 1;

function Cesty3(K: integer): integer;
  {počet různých cest z města K do města N --- rekurze
   s použitím globálního pomocného pole C pro uložení
   již známých hodnot}
  begin
  if C[K] >= 0 then          {hodnota je již známa z dřívějška}
    Cesty3 := C[K]
  else
```



```
  begin
  C[K] := 0;
  if X[K] > 0 then C[K] := C[K] + Cesty3(X[K]);
  if Y[K] > 0 then C[K] := C[K] + Cesty3(Y[K]);
  Cesty3 := C[K]
  end
end;  {function Cesty3}
```

Také tuto úlohu je možné řešit jednoduchým lineárním algoritmem pomocí cyklu. Zbytečnou rekurzi odstraníme stejným obratem jako v případě Fibonacciho čísel. Postup je založen na využití skutečnosti, že každá silnice vede vždy do města s vyšším číslem. Můžeme proto počítat pro všechna města v pořadí $N, N-1, \ldots, 1$, kolik z nich vede různých cest do města $N$. Získané hodnoty si budeme ukládat do pomocného pole $C$, které má stejný význam a obsah jako u funkce *Cesty3*. Počet cest vedoucích z města $K$ do města $N$ je roven součtu počtů cest vedoucích z měst $X[K]$ a $Y[K]$ do $N$. Hodnotu $C[K]$ tedy počítáme jako součet hodnot $C[X[K]]$ a $C[Y[K]]$. Vzhledem k podmínkám $X[K] > K$ a $Y[K] > K$ a ke zvolenému pořadí výpočtu hodnot prvků pole $C$ od vyšších indexů k nižším budou v okamžiku, kdy počítáme $C[K]$, údaje $C[X[K]]$ a $C[Y[K]]$ již známy. Můžeme je proto jednoduše použít. Algoritmus zapíšeme ve tvaru funkce *Cesty4*.

```
function Cesty4: integer;
  {počet různých cest z města 1 do města N - cyklem
   s použitím pomocného pole C pro uložení již známých
   hodnot}
  var C: array[1..N] of integer; {počet cest vedoucích do N}
      K: integer;
  begin
  C[N] := 1;
  for K:=N-1 downto 1 do
    begin
    C[K] := 0;
    if X[K] > 0 then C[K] := C[K] + C[X[K]];
    if Y[K] > 0 then C[K] := C[K] + C[Y[K]]
    end;
  Cesty4 := C[1]
  end;  {function Cesty4}
```

Všimněte si, že na rozdíl od výpočtu Fibonacciho čísel pomocí cyklu (funkce *Fib4* v kap. 13.1) jsme zde museli použít pomocné pole velikosti $N$. K určení každého nově počítaného Fibonacciho čísla nám stačilo znát dvě bezprostředně předcházející hodnoty, k jejich uložení jsme proto vystačili s dvěma proměnnými. V případě silniční sítě se nová hodnota počítá na základě nejvýše dvou již známých hodnot, ne nutně však bezprostředně předcházejících. Musíme si proto udržovat všechny spočítané hodnoty v pomocném poli, neboť předem nevíme, které z nich budeme při výpočtu ještě potřebovat.

Zápis funkce *Cesty4* je ještě možné zkrátit a zjednodušit, což ovšem už nic nezmění na lineární časové složitosti algoritmu. Použijeme k tomu malý technický trik. Jestliže pole $C$ doplníme o jeden pomocný prvek s indexem 0 a tomuto prvku nastálo přiřadíme nulovou hodnotu, nebudeme muset v algoritmu zvlášť testovat, která silnice z města $K$ vede a která ne (tj. zda $X[K]$ nebo $Y[K]$ není nula). Úpravou tak dostáváme funkci *Cesty5*.

```
function Cesty5: integer;
  {počet různých cest z města 1 do města N - cyklem,
   s použitím pomocného pole C pro uložení již známých
   hodnot}
  var C: array[0..N] of integer; {počet cest vedoucích do N}
      K: integer;
  begin
  C[0] := 0;              {umělá hodnota z technických důvodů}
  C[N] := 1;
  for K:=N-1 downto 1 do
    C[K] := C[X[K]] + C[Y[K]];
  Cesty5 := C[1]
  end;  {function Cesty5}
```

## CVIČENÍ

**1.** Definujme si rekurentním předpisem posloupnost čísel podobnou Fibonacciho číslům:

$$F(1) = 1,$$
$$F(N) = F(N-1) + F(N/2) \quad \text{pro } N > 1.$$



Znak „/“ v definici představuje operaci celočíselného dělení. Napište funkci, která spočítá hodnotu $F(K)$ pro zadané číslo $K$.

**2.** Napište program na výpočet prvních 100 Fibonacciho čísel. Dejte pozor na aritmetické přetečení při výpočtu. K uložení Fibonacciho čísel vám nepostačí ani standardní 32bitový celočíselný datový typ (jako je třeba `longint` v Turbo Pascalu), do proměnné tohoto typu se vejde nejvýše 46. Fibonacciho číslo.

# 14 DYNAMICKÉ PROGRAMOVÁNÍ

Dynamické programování je jednou z myšlenkově poněkud náročnějších metod, jak navrhovat efektivní algoritmy. Můžeme ho použít pro určitou třídu úloh, u kterých je možné výhodně sestavovat řešení úlohy z již známých řešení dílčích podúloh. To nám trochu připomíná metodu rozděl a panuj (viz kap. 10). Zásadní odlišnost však spočívá v tom, že dílčí podúlohy téže úrovně zde neřešíme nezávisle na sobě v libovolném pořadí, jako tomu bylo u metody rozděl a panuj, ale řešíme je všechny najednou s využitím společně sdílených znalostí o výsledcích podúloh z nižších úrovní.

Úlohy, které lze řešit metodou dynamického programování, vypadají obvykle na první pohled tak, že nás nenapadne nic lepšího než zkoušení všech možností pomocí backtrackingu (viz kap. 8.2). Na rozdíl od backtrackingu, který bývá velmi pomalý a dovoluje řešit v rozumném čase pouze malé úlohy (vede zpravidla k programům s exponenciální časovou složitostí), bývá algoritmus založený na metodě dynamického programování nesrovnatelně rychlejší. Výsledný program má vždy polynomiální časovou složitost, nejčastěji řádu $O(N^3)$. Je tedy dobře použitelný i pro řešení velkých úloh. Podstatného zrychlení výpočtu se dosahuje za cenu použití pomocné paměti k ukládání opakovaně využívaných mezivýsledků. Paměťová náročnost programů založených na této metodě bývá o řád nižší než náročnost časová, tzn. typicky $O(N^2)$.

Základní idea dynamického programování je velmi blízká technice zvyšování efektivity rekurzivních algoritmů, kterou jsme si vyložili v kap. 13. Tam šlo o to, že jsme rekurzivní algoritmus doplnili pomocným polem sloužícím k ukládání všech již jednou spočtených hodnot. Jestliže jsme později nějakou hodnotu potřebovali, počítali jsme ji pomocí rekurzivního volání jedině tehdy, pokud ještě nebyla z dřívějška spočtena a uložena v poli. Celá úloha se při řešení rekurzivně rozkládala na podúlohy, každá z nich opět na menší podúlohy atd. až po dosažení nějakých elementárních přímo řešitelných úloh. Všechny dílčí podúlohy bylo sice možné řešit nezávisle na sobě, pokud by se však některé z nich shodovaly, představovalo by to velké množství práce prováděné zbytečně vícekrát. Použití pomocného pole takovým zbytečným opakovaným výpočtům zabránilo a tím zásadně zrychlilo výpočet.

V případě dynamického programování jsme opět v situaci, že k řešení



úlohy potřebujeme vyřešit nějaké její podúlohy, ty mají zase své dílčí podúlohy atd. Rovněž platí, že mnohé dílčí podúlohy se v této hierarchii opakují. K řešení však tentokrát nepoužijeme rekurzivní algoritmus rozkládající úlohu postupně shora dolů, ale naopak budeme postupovat zdola nahoru. Nejprve vyřešíme všechny elementární podúlohy a jejich výsledky si zaznamenáme do pole. Pomocí nich pak vyřešíme vyšší podúlohy a výsledky opět zapíšeme do pole. Stejně postupujeme dál, řešíme stále větší a větší dílčí podúlohy. K tomu vždy využíváme v poli uložené výsledky již vyřešených nižších podúloh. Mnohé z těchto uložených výsledků budou využity vícekrát, jiné zase třeba ani jednou. Takto pokračujeme až do vyřešení celé původní úlohy.

Všimněte si, že uvedené myšlence dynamického programování přesně odpovídá i postup použitý při řešení úloh o Fibonacciho číslech a o silniční síti v kap. 13. Efektivní postup řešení zdola nahoru je tam uplatněn ve funkcích *Fib4*, *Cesty4* a *Cesty5*. Tomuto „jednorozměrnému případu" se však obvykle neříká dynamické programování, pojem dynamické programování bývá zvykem užívat až v situaci, kdy pole uchovávající informace je alespoň dvojrozměrné.

## 14.1 Úloha o násobení matic

Metodu dynamického programování si nyní ukážeme na několika konkrétních příkladech. Začneme úlohou, která se v této souvislosti nejčastěji objevuje v učebnicích algoritmů. Máme dáno $N$ obdélníkových matic $A_1, A_2, \ldots, A_N$ a chceme spočítat jejich součin $A_1 A_2 \ldots A_N$. Naším úkolem je vhodným uzávorkováním tohoto součinu minimalizovat počet násobení jednotlivých prvků matic. Nemáme tedy přímo počítat vlastní součin matic, ale máme zjistit, jaký je minimální počet násobení prvků potřebný k výpočtu uvedeného součinu matic.

Bude dobré učinit k úloze nejprve několik poznámek. Především si připomeneme, že matice je obdélníkové schéma čísel, například reálných. O matici s $p$ řádky a $q$ sloupci říkáme, že je tvaru $p \times q$. Pro matice $B$, $C$ má součin $BC$ smysl pouze tehdy, pokud počet sloupců matice $B$ je roven počtu řádků matice $C$. Je-li matice $B$ tvaru $p \times q$ a matice $C$ tvaru $q \times r$, můžeme spočítat součin $BC$. Bude jím matice tvaru $p \times r$, jejíž prvek $a_{i,j}$ se určí jako součet výrazů $b_{i,k}c_{k,j}$ pro všechna $k$ od 1 do $q$. Výsledná matice má $pr$ prvků, k výpočtu každého z nich je třeba provést $q$

násobení, takže celkově se při jejím výpočtu vykoná $pqr$ násobení prvků. V naší úloze tedy musí být každá matice $A_i$ tvaru $p_{i-1} \times p_i$ (pro $i = 1, \ldots, N$). Vstupem programu pak bude posloupnost kladných celých čísel $p_0, p_1, \ldots, p_N$ popisující rozměry násobených matic. Na konkrétních hodnotách prvků matic výsledek řešení vůbec nezávisí.

Operace násobení matic je asociativní, takže zápis $A_1 A_2 \ldots A_N$ můžeme libovolně uzávorkovat, aniž bychom tím ovlivnili výsledek násobení. Na volbě uzávorkování však může záviset počet vykonaných násobení prvků. To si snadno předvedeme na konkrétním příkladu. Nechť $N = 3$ a počítáme součin tří matic $A_1 A_2 A_3$. Matice $A_1$ má rozměry $2 \times 3$, $A_2$ je tvaru $3 \times 2$, $A_3$ má velikost $2 \times 5$. Při výpočtu $A_1 A_2$ provedeme 12 násobení a získáme matici tvaru $2 \times 2$, k výpočtu $(A_1 A_2) A_3$ bude zapotřebí $12 + 20 = 32$ násobení. V případě jiného uzávorkování spočítáme napřed matici $A_2 A_3$ tvaru $3 \times 5$ pomocí 30 násobení prvků, k celkovému výpočtu $A_1 (A_2 A_3)$ bude celkem provedeno $30 + 30 = 60$ násobení.

Úlohu by bylo možné řešit systematickým zkoumáním všech možných uzávorkování daného součinu matic. Naprogramovat takové řešení pomocí rekurzivní funkce je celkem snadné, výsledný program je však dosti neefektivní. Jenom pro ilustraci si ukážeme, jak by takové řešení mohlo vypadat. Předpokládejme, že v globálním poli $P[0..N]$ jsou uloženy rozměry násobených matic podle zadání. Celou úlohu bude řešit rekurzivní funkce *PocetNasobeni*, která jako vstupní parametry $K$, $L$ dostane dvě pořadová čísla matic taková, že $1 \leqq K \leqq L \leqq N$. Funkce určí minimální počet násobení prvků potřebný k výpočtu součinu $A_K A_{K+1} \ldots A_L$. Funkce svůj výsledek spočítá tak, že vyzkouší všechny možnosti, které z naznačených $L - K$ násobení matic lze vykonat jako poslední. Pro každou z těchto možností určí počet provedených násobení prvků a z takto získaných počtů vybere minimum. Při každé volbě místa posledního násobení matic určíme celkový počet provedených násobení prvků jako součet tří hodnot: minimálního počtu násobení prvků nutných pro výpočet úseku vlevo od místa posledního násobení matic, minimálního počtu násobení prvků nutných pro výpočet pravého úseku a počtu násobení prvků vykonaných při posledním násobení matic. Minimální počet násobení prvků potřebný pro zpracování jednoho dílčího úseku matic spočítáme vždy pomocí rekurzivního volání funkce *PocetNasobeni*. Naznačený postup si ukážeme v Pascalu.



```
program Soucin_Matic_Rekurzi;
{Pomocí rekurzivní funkce určuje minimální počet násobení
potřebný k vynásobení N obdélníkových matic. Vstupem
programu jsou rozměry matic.}

const MaxN = 100;   {maximální počet násobených matic}

var   P: array[0..MaxN] of integer;
      {rozměry matic -> i-tá matice má velikost P[i-1]xP[i]}
      i: integer;

function PocetNasobeni(K,L: integer):integer;
{parametry K, L vymezují sledovaný úsek v řadě násobených
 matic - funkce počítá součin matic od K-té po L-tou včetně}
var Poc: integer;          {k výpočtu počtu násobení}
    Min: integer;          {minimální počet násobení}
    i: integer;
begin
  if K = L then
    PocetNasobeni:=0       {úsek tvořen jedinou maticí,
                            žádné násobení se neprovádí}
  else
    begin
    Min:=maxint;
    for i:=K to L-1 do  {"poslední" násobení je
                         za i-tou maticí}
      begin
      Poc:=PocetNasobeni(K,i) + PocetNasobeni(i+1,L)
                                          {násobení v úsecích}
           + P[K-1] * P[i] * P[L];        {poslední násobení}
      if Poc < Min then Min:=Poc
      end;
    PocetNasobeni:=Min
    end
end;  {function PocetNasobeni}

begin
writeln('Zadejte rozměry násobených matic:');
i:=0;
```

```
while not eof do
  begin read(P[i]); i:=i+1 end;
writeln('Minimální počet násobení: ', PocetNasobeni(1,i-1))
end.
```

Lepší řešení naší úlohy je založeno na metodě dynamického programování. Definujme si hodnoty $M_{I,J}$ jako minimální počet násobení prvků potřebný k určení součinu $I$ po sobě jdoucích matic počínaje maticí $A_J$, tedy součinu $A_J A_{J+1} \ldots A_{J+I-1}$. Údaj $M_{I,J}$ má smysl pro všechna $I$ od 1 do $N$, pro každé takové $I$ potom pro $J$ od 1 do $N - I + 1$. Zřejmě platí $M_{1,J} = 0$ pro každé $J$, $M_{N,1}$ je hledaným výsledkem úlohy.

Hodnoty $M_{I,J}$ budeme počítat postupně podle rostoucího indexu $I$ a budeme je ukládat do pomocné matice $M[1..N, 1..N]$. Matici $M$ tedy budeme zaplňovat po řádcích. První řádek $M$ obsadíme samými nulami, v posledním řádku pak nalezneme v prvku $M[N, 1]$ výsledek. Hodnoty $I$-tého řádku matice $M$ spočítáme vždy s využitím zaznamenaných hodnot z řádků 1, 2, ..., $I - 1$. Zbývá ukázat, jak spočítáme $M_{I,J}$ pro nějaká pevně zvolená $I$, $J$ z přípustného rozmezí ($I > 1$). Hodnota $M_{I,J}$ udává minimální počet násobení prvků provedených při výpočtu $A_J A_{J+1} \ldots A_{J+I-1}$. Stejně jako v případě předchozího rekurzivního řešení úlohy budeme uvažovat, které z těchto násobení matic se provede jako poslední, a ze všech možností vybereme tu s minimálním počtem násobení prvků. Od této úvahy již vede přímá cesta k předpisu

$$M_{I,J} = \min(M_{K,J} + M_{I-K,J+K} + p_{J-1} p_{J+K-1} p_{J+I-1})$$

Ve vzorci uvedené minimum se počítá přes všechna $K$ od 1 do $I - 1$, přičemž toto $K$ představuje počet matic ležících v součinu $A_J A_{J+1} \ldots$ $\ldots A_{J+I-1}$ vlevo od naposledy prováděného násobení.

Časové i paměťové nároky uvedeného algoritmu jsou zřejmé z popisu. Využívá se pomocné pole $M$ rozměrů $N \times N$ (z něhož se část nevyužije), tedy paměť velikosti $O(N^2)$. Celé řešení úlohy vlastně spočívá v postupném výpočtu hodnot prvků tohoto pole. Počet prvků je úměrný $N^2$, k výpočtu každého z nich se vykoná řádově $N$ operací, takže celková časová složitost algoritmu vychází $O(N^3)$.

Ukážeme si kompletní řešení úlohy v Pascalu. Program předpokládá, že na vstupu dostane zadánu posloupnost rozměrů násobených matic $p_0$, $p_1$, ..., $p_N$ tak, jak je uvedeno v zadání úlohy. Pro zjednodušení opět



předpokládáme, že všechna vstupní data jsou zadána správně a že poslední ze vstupních údajů je bezprostředně následován příznakem konce souboru (tak, jak je to možné zadat v Turbo Pascalu).

```
program Soucin_Matic_Dynamicky;
{Metodou dynamického programování určuje minimální počet
násobení potřebný k vynásobení N obdélníkových matic.
Vstupem programu jsou rozměry matic.}

const MaxN = 100;   {maximální počet násobených matic}

var P: array[0..MaxN] of integer;
     {rozměry matic -> i-tá matice má velikost P[i-1]xP[i]}
    M: array[1..MaxN,1..MaxN] of integer;
    N: integer;              {počet matic}
    Poc: integer;            {k výpočtu počtu násobení}
    Min: integer;            {minimální počet násobení}
    i, j, k: integer;

begin
writeln('Zadejte rozměry násobených matic:');
i:=0;
while not eof do
  begin read(P[i]); i:=i+1 end;
N:=i-1;                      {počet matic}

for j:=1 to N do M[1,j]:=0;
for i:=2 to N do             {řádky matice M}
  for j:=1 to N-i+1 do
    begin
    Min:=maxint;
    for k:=1 to i-1 do
      begin
      Poc:=M[K,j] + M[i-k,j+k]          {násobení v úsecích}
           + P[j-1] * P[j+k-1] * P[j+i-1];
                                        {poslední násobení}
      if Poc < Min then Min:=Poc
      end;
    M[i,j]:=Min
    end;
```

```
writeln('Minimální počet násobení: ', M[N,1])
end.
```

## 14.2 Úloha o triangulaci mnohoúhelníku

Druhá úloha, na níž si metodu dynamického programování předvedeme, byla použita před několika lety jako soutěžní úloha celostátního kola matematické olympiády — kategorie P (programování). Její stručné řešení proto můžete nalézt v příslušné ročence MO (39. ročník), podrobněji je rozebrána také v knize [14].

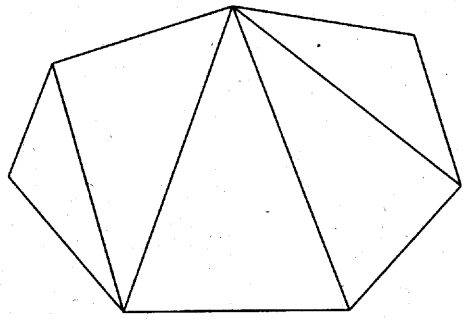
Konvexní $N$-úhelník $A_1A_2\ldots A_N$ ($N > 3$) je zadán kartézskými souřadnicemi svých vrcholů v rovině. Naším úkolem je určit velikost jeho minimální triangulace. Musíme si ovšem nejdříve vysvětlit, co budeme rozumět pod pojmem minimální triangulace $N$-úhelníku.

**Diagonálou** konvexního $N$-úhelníku nazýváme každou úsečku spojující dva jeho různé vrcholy takové, že spolu nesousedí na obvodu (tj. nejsou spojeny hranou $N$-úhelníku). Z každého vrcholu $N$-úhelníku vychází přesně $N-3$ diagonál. **Triangulací** konvexního $N$-úhelníku označujeme takový soubor jeho navzájem se neprotínajících diagonál, které rozdělují plochu $N$-úhelníku na samé trojúhelníky. Každý konvexní $N$-úhelník s více než třemi vrcholy má více různých triangulací, každá z nich je tvořena $N-3$ diagonálami. Součet délek všech diagonál, které tvoří jednu konkrétní triangulaci $N$-úhelníku, nazýváme **velikost** této triangulace. **Minimální triangulace** konvexního $N$-úhelníku je ta z jeho triangulací, která má minimální velikost. Některé $N$-úhelníky mohou mít více různých minimálních triangulací. A právě velikost minimální triangulace daného konvexního $N$-úhelníku máme v naší úloze nalézt.

Přesnou matematickou formulaci úlohy si můžeme nahradit velmi názornou představou: Naším úkolem je rozřezat daný konvexní mnohoúhelník na trojúhelníky úsečkami spojujícími jeho vrcholy tak, aby součet délek těchto úseček byl minimální.

K řešení úlohy by bylo možné použít stejně jako v předchozí úloze metodu backtrackingu. Postupným umisťováním jednotlivých diagonál všemi možnými způsoby bychom mohli vygenerovat všechny triangulace daného mnohoúhelníku, určit velikost každé z nich a z těchto velikostí





*Obr. 30* Triangulace konvexního mnohoúhelníku

vzít minimum. Detailnějším řešením úlohy touto neefektivní cestou se tentokrát již nebudeme zabývat a raději si hned ukážeme, jak úlohu vyřešíme dynamickým programováním.

Představme si, že do daného $N$-úhelníku zakreslíme nějakou jeho minimální triangulaci. Jestliže nyní zvolíme jednu libovolnou diagonálu z této triangulace, diagonála nám $N$-úhelník rozděluje na dva menší mnohoúhelníky. Všimněte si, že v každém z nich je zakreslena jeho minimální triangulace. Minimální triangulace $N$-úhelníku se tedy skládá z minimálních triangulací dílčích mnohoúhelníků, na které ho můžeme rozložit (a ještě obsahuje dělící diagonálu). Z tohoto pozorování vychází celý algoritmus. Jak již víme, nebudeme při výpočtu postupovat rekurzivním rozkladem mnohoúhelníku shora dolů. Začneme naopak zdola od minimálních triangulací menších dílčích mnohoúhelníků. Z nich budeme postupně sestavovat minimální triangulace větších a větších mnohoúhelníků, až najdeme minimální triangulaci celého daného $N$-úhelníku.

Definujeme si hodnotu $M_{I,J}$ jako velikost minimální triangulace mnohoúhelníku tvořeného $I$ po sobě jdoucími vrcholy na obvodu daného $N$-úhelníku počínaje vrcholem $A_J$, zvětšenou navíc o velikost úsečky $A_JA_{J+I-1}$. Je to tedy velikost minimální triangulace mnohoúhelníku $A_JA_{J+1}\ldots A_{J+I-1}$ zvětšená o délku úsečky $A_JA_{J+I-1}$. Číslování vrcholů přitom budeme chápat cyklicky, po vrcholu $A_N$ následuje opět vrchol $A_1$. Zápis $A_{N+1}$ připustíme jako správný a bude znamenat totéž co $A_1$, $A_{N+2}$ je totéž jako $A_2$ atd. Údaj $M_{I,J}$ má smysl pro všechna $I$ od 3 do $N-1$ a pro všechna $J$ od 1 do $N$. Triangulace trojúhelníku je nulová,

takže hodnota $M_{3,J}$ bude rovna přímo délce úsečky $A_JA_{J+2}$ pro libovolné $J$. Hodnoty $M_{N-1,J}$ představují velikost minimální triangulace celého daného $N$-úhelníku takové, že obsahuje diagonálu $A_JA_{J+N-2}$. Výsledná minimální triangulace daného $N$-úhelníku musí obsahovat některou z hran typu $A_JA_{J+2}$, takže mezi hodnotami $M_{N-1,J}$ rozhodně najdeme řešení úlohy (jako minimum z hodnot $M_{N-1,J}$ pro všechna $J$ od 1 do $N$).

Hodnoty $M_{I,J}$ budeme počítat postupně podle rostoucího indexu $I$. Budeme si je ukládat do pomocné matice $M$ velikosti $M[2..N-1, 1..N]$. Matici $M$ tedy budeme stejně jako v úloze z kap. 14.1 zaplňovat po řádcích. Hodnoty $M_{3,J}$ spočítáme přímo jako vzdálenosti vrcholů daného mnohoúhelníku, $M_{3,J}$ je rovno délce úsečky $A_JA_{J+2}$ pro každé $J$. Hodnoty $I$-tého řádku matice $M$ pro $I > 3$ pak budeme počítat vždy pomocí hodnot uložených v řádcích $3, 4, \ldots, I-1$. Zbývá ukázat, jak spočítáme $M_{I,J}$ pro nějaká pevně zvolená $I$, $J$ taková, že $3 < I < N$, $1 \leqq J \leqq N$. Každá triangulace $I$-úhelníku $A_JA_{J+1}\ldots A_{J+I-1}$ musí obsahovat dvojici úseček $A_JA_{J+K}$, $A_{J+K}A_{J+I-1}$ pro nějaký index $K$ z rozmezí od 1 do $I-2$. Triangulace totiž musí dělit plochu mnohoúhelníku na samé trojúhelníky, jeden z takto vzniklých trojúhelníků má jako jednu svoji stranu úsečku $A_JA_{J+I-1}$ a jeho třetím vrcholem musí být právě některý bod $A_{J+K}$. Při výpočtu $M_{I,J}$ budeme proto postupně zkoumat minimální triangulace obsahující dvojici úseček $A_JA_{J+K}$, $A_{J+K}A_{J+I-1}$ pro všechna $K$ od 1 do $I-2$, což jsou všechny možné triangulace mnohoúhelníku $A_JA_{J+1}\ldots A_{J+I-1}$. Velikost každé takové triangulace snadno spočítáme jako součet již známých hodnot $M_{K+1,J}$ a $M_{I-K,J+K}$. Velikost minimální triangulace mnohoúhelníku $A_JA_{J+1}\ldots A_{J+I-1}$ je rovna minimu z těchto hodnot, nová hodnota $M_{I,J}$ je větší ještě o délku úsečky $A_JA_{J+I-1}$. Spočítáme ji tedy podle předpisu:

$$M_{I,J} = \min(M_{K+1,J} + M_{I-K,J+K}) + \text{délka}(A_JA_{J+I-1})$$

Ve vzorci uvedené minimum se počítá přes všechna $K$ od 1 do $I-2$. Zbývá vyřešit již jen malou technickou záležitost. Pro mezní hodnoty $K$, tj. hodnoty $K = 1$ a $K = I-2$, ve vzorci vystupují veličiny $M_{2,J}$, $M_{2,J+K}$. Ty ale nejsou vůbec definovány a nemají věcně smysl, neboť mnohoúhelník se dvěma vrcholy neexistuje. Pokud si podrobněji rozeberete, jak ve vzorci vznikly, zjistíte, že to není chyba a že ke správnému fungování algoritmu je třeba položit $M_{2,J} = 0$ pro všechna $J$ od 1 do $N$.



Časovou a paměťovou složitost algoritmu odvodíme velmi snadno. K výpočtu se využívá pomocné pole $M$ o zhruba $N^2$ prvcích. Celý výpočet spočívá v postupném určení hodnot všech $N^2$ prvků tohoto pole. Ke stanovení každé z těchto hodnot je zapotřebí provést řádově N elementárních operací. Algoritmus má tedy paměťovou složitost $O(N^2)$ a časovou složitost $O(N^3)$.

Při psaní programu musíme ještě dořešit některé detaily, kterým jsme v rozboru algoritmu nevěnovali pozornost. Vrcholy $N$-úhelníku jsou zadány svými kartézskými souřadnicemi, ale celý algoritmus pracuje pouze se vzdálenostmi dvojic vrcholů. Budeme proto potřebovat funkci pro výpočet délky úsečky v rovině. Dále budeme potřebovat funkci pro přepočítávání indexů vrcholů mnohoúhelníku modulo $N$, která zajistí ztotožnění vrcholu $A_{N+1}$ s vrcholem $A_1$, vrcholu $A_{N+2}$ s vrcholem $A_2$ atd. Vstupem programu bude nejprve údaj o počtu vrcholů zkoumaného mnohoúhelníku a potom pro každý vrchol dvojice jeho kartézských souřadnic. Vrcholy jsou zadány po řadě, jak za sebou následují na obvodu mnohoúhelníku. Následující program pro jednoduchost nekontroluje správnost vstupních dat, tj. nekontroluje, zda zadaný počet vrcholů mnohoúhelníku je větší než 3 a zda vrcholy v daném pořadí tvoří konvexní mnohoúhelník. Doplnění těchto testů by nebylo obtížné.

```
program MinimalniTriangulace1;
{Pro daný mnohoúhelník zadaný kartézskými souřadnicemi
 vrcholů v rovině určí velikost minimální triangulace}

const MaxN = 20;            {max. počet vrcholů mnohoúhelníku}
      MaxNMinus1 = 19;      { = MaxN - 1 }
type TypVrcholu = record X,Y: real end;

var Souradnice: array[1..MaxN] of TypVrcholu;
                                    {uložení souřadnic vrcholů}
    N: integer;             {počet vrcholů mnohoúhelníku}
    M: array[2..MaxNMinus1,1..MaxN] of real;
    Min, Triang: real;      {k určení minimální triangulace}
    I, J, K: integer;

function Vzdal(K, L: integer): real;
  {počítá vzdálenost vrcholů s indexy K a L v rovině
   pomocí Pythagorovy věty}
```

```
  begin
  Vzdal := sqrt(sqr(Souradnice[K].X - Souradnice[L].X)
             + sqr(Souradnice[K].Y - Souradnice[L].Y))
  end;  {function Vzdal}

function Index(K: integer): integer;
  {přepočítává index vrcholu, aby byl z rozmezí od 1 do N;
   předp.: vstupní hodnota K je z rozmezí od 1 do 2.N}
  begin
  if K <= N then Index := K
            else Index := K-N
  end;  {function Index}

begin
{Načtení vstupních dat:}
write('Počet vrcholů mnohoúhelníku (N > 3): ');
readln(N);
writeln('Souřadnice jednotlivých vrcholů v pořadí po obvodu:');
for I:=1 to N do read(Souradnice[I].X, Souradnice[I].Y);

{Přímé určení hodnot M[2,J] a M[3,J]:}
for J:=1 to N do
  begin
  M[2,J] := 0;
  M[3,J] := Vzdal(J,Index(J+2))
  end;

{Zaplnění tabulky M:}
for I:=4 to N-1 do
  for J:=1 to N do
    begin  {určení M[I,J] pro I>3}
    Min := M[I-1,Index(J+1)];  {případ K=1}
    for K:=2 to I-2 do
      begin
      Triang := M[K+1,J] + M[I-K,Index(J+K)];
      if Triang < Min then
        Min := Triang;
      end;
    M[I,J] := Min + Vzdal(J,Index(J+I-1));
    end;
```



```
čení a výpis výsledku:}
 := M[N-1,1];
 J:=2 to N do
if M[N-1,J] < Min then
  Min :=M [N-1,J];
iteln;
iteln('Velikost minimální triangulace: ', Min:10:5)
d.
```

Zamysleme se nyní, co by se v řešení úlohy změnilo, kdybychom chtěli určit nejen velikost minimální triangulace, ale i seznam diagonál, které ji tvoří. Hodnoty $M_{I,J}$ jsou postupně počítány od nižších indexů $I$ k vyšším a předem nevíme, které z nich se uplatní ve výsledném řešení. Musíme si proto pro jistotu s každým spočítaným údajem $M_{I,J}$ zároveň zapamatovat, jak jsme ho získali. Mnohé z takto zapamatovaných informací se sice nikdy na nic nepoužijí, ale předem nevíme, které potřebovat budeme. Definujeme si hodnoty $D_{I,J}$ s následujícím významem: v předpisu pro výpočet $M_{I,J}$ se nabývá minima pro hodnotu $K = D_{I,J}$. Údaj $D_{I,J}$ nám tedy říká, díky kterému vrcholu mnohoúhelníku jsme získali příslušnou hodnotu $M_{I,J}$. Zajímat nás budou pouze $D_{I,J}$ pro $I > 3$, určíme je velmi snadno vždy zároveň s výpočtem $M_{I,J}$.

Po stanovení všech hodnot $M_{I,J}$ (a také $D_{I,J}$) vybereme minimum z čísel $M_{N-1,J}$ pro $J$ od 1 do $N$ — nechť je to $M_{N-1,X}$. To představuje velikost minimální triangulace. Zbývá nám ještě zrekonstruovat minimální triangulaci samotnou. Jistě obsahuje diagonálu $A_X A_{X+N+2}$, to plyne přímo z minimality čísla $M_{N-1,X}$. Díky $D_{N-1,X}$ víme, že obsahuje také úsečky vedoucí z vrcholů $A_X$ a $A_{X+N+2}$ do vrcholu s indexem $X + D_{N-1,X}$ (pokud to jsou diagonály a ne strany mnohoúhelníku, což se někdy může stát). Posledně uvedené dvě úsečky oddělují z celého $N$-úhelníku každá jeden menší mnohoúhelník (jsou-li to diagonály). Minimální triangulace těchto dílčích mnohoúhelníků najdeme obdobným způsobem pomocí jim odpovídajících hodnot $D$. Stejně postupujeme stále dál až do rozložení daného $N$-úhelníku na samé trojúhelníky a tím nalezení všech diagonál tvořících minimální triangulaci.

Asymptotická časová a paměťová složitost celého řešení se touto úpravou nezměnila. Stanovení hodnot $D_{I,J}$ probíhá zároveň s výpočtem odpovídajících $M_{I,J}$. Dodatečně přidaná závěrečná etapa má sice charakter rekurzivního rozkladu, ten však není obecným průchodem do

hloubky s exponenciální časovou složitostí, ale je připravenými údaji $D$ přímo řízen a má díky tomu lineární časovou složitost. V každém kroku výpočtu se vytvářený soubor diagonál doplní o jednu úsečku, přičemž celá triangulace je tvořena $N - 3$ diagonálami.

Sestavení výsledné minimální triangulace naprogramujeme pomocí rekurzivní funkce *Diagonala*. Parametry funkce udávají počet vrcholů a index prvního vrcholu zkoumaného mnohoúhelníku. Funkce přímo vytiskne jednu diagonálu náležející do minimální triangulace a podle potřeby zajistí rekurzivním voláním výpis dalších diagonál patřících do triangulace.

---

```
program MinimalniTriangulace2;
{Pro daný mnohoúhelník zadaný kartézskými souřadnicemi
 vrcholů v rovině určí velikost minimální triangulace
 a vypíše také soubor diagonál, který minimální
 triangulaci tvoří}

const MaxN = 20;            {max. počet vrcholů mnohoúhelníku}
      MaxNMinus1 = 19;      { = MaxN - 1 }
type TypVrcholu = record X,Y: real end;

var Souradnice: array[1..MaxN] of TypVrcholu;
                                {uložení souřadnic vrcholů}
    N: integer;             {počet vrcholů mnohoúhelníku}
    M: array[2..MaxNMinus1,1..MaxN] of real;
    D: array[4..MaxNMinus1,1..MaxN] of integer;
    Min, Triang: real;      {k určení minimální triangulace}
    X: integer;             {pro práci s hodnotami D}
    I, J, K: integer;

function Vzdal(K, L: integer): real;
  {počítá vzdálenost vrcholů s indexy K a L v rovině
   pomocí Pythagorovy věty}
  begin
  Vzdal := sqrt(sqr(Souradnice[K].X - Souradnice[L].X)
             + sqr(Souradnice[K].Y - Souradnice[L].Y))
  end;  {function Vzdal}

function Index(K: integer): integer;
  {přepočítává index vrcholu, aby byl z rozmezí od 1 do N;
```



```
   dp.: vstupní hodnota K je z rozmezí od 1 do 2.N}
  in
  K <= N then Index := K
        else Index := K-N
  ;  {function Index}

  edure Diagonala(I, J: integer);
  ypis minimální triangulace
   I-úhelník počínaje vrcholem A[J]}
  gin
  riteln(J:5, Index(J+I-1):5);
  f I > 3 then
   begin
   if D[I,J] > 1 then Diagonala(D[I,J]+1,J);
   if D[I,J] < I-2 then Diagonala(I-D[I,J],Index(J+D[I,J]))
   end
  nd;  {procedure Diagonala}

  gin
  ačtení vstupních dat:}
  ite('Počet vrcholů mnohoúhelníku (N > 3): ');
  eadln(N);
  riteln('Souřadnice jednotlivých vrcholů v pořadí po obvodu:');
  or I:=1 to N do read(Souradnice[I].X, Souradnice[I].Y);

 {Přímé určení hodnot M[2,J] a M[3,J]:}
 for J:=1 to N do
   begin
   M[2,J] := 0;
   M[3,J] := Vzdal(J,Index(J+2))
   end;

 {Zaplnění tabulky M a D:}
 for I:=4 to N-1 do
   for J:=1 to N do
     begin  {určení M[I,J] pro I>3}
     Min := M[I-1,Index(J+1)];  {případ K=1}
     X := 1;
     for K:=2 to I-2 do
       begin
       Triang := M[K+1,J] + M[I-K,Index(J+K)];
```

```
      if Triang < Min then
        begin
        Min := Triang;
        X := K
        end
      end;
    M[I,J] := Min + Vzdal(J,Index(J+I-1));
    D[I,J] := X
    end;

{Určení a výpis výsledku:}
Min := M[N-1,1];
X := 1;
for J:=2 to N do
  if M[N-1,J] < Min then
    begin
    Min := M[N-1,J];
    X := J
    end;
writeln;
writeln('Velikost minimální triangulace: ', Min:10:5);
writeln('Diagonály tvořící minimální triangulaci:');
Diagonala(N-1,X)   {výpis diagonál pomocí údajů v poli D}
end.
```

## 14.3 Úloha o cestování

Metodu dynamického programování si předvedeme ještě na jedné rozsáhlejší úloze, která tematicky čerpá z oblasti cestování. Po stránce algoritmické je tato úloha blízká grafovým úlohám, které jsme řešili v kap. 9. Jde v ní totiž o nalezení nejlepší možné cesty v daném grafu s dosti speciálním stanovením kritéria, podle něhož cestu vybíráme. Úloha pochází z 5. mezinárodní olympiády středoškoláků v informatice, kde byla zadána jako nejtěžší a nejvíce bodovaná soutěžní úloha. Jenom několik málo účastníků soutěže znalo a umělo využít princip dynamického programování a pomocí něho napsat velmi efektivní program, který prohledal i rozsáhlý graf v krátkém čase. Většina řešitelů nepřišla na nic jiného než na triviální řešení úlohy pomocí backtrackingu. Takový program počítá



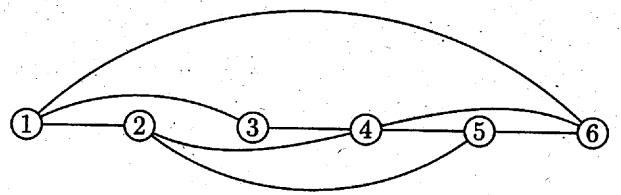
v principu správně, ale pro komplikovanější grafy je neúnosně pomalý prakticky nepoužitelný.

Vlastní text úlohy si stylisticky poněkud upravíme, věcně se však bude lišit od zadání soutěžní úlohy na olympiádě. V jisté zemi je $N$ velkých měst. Města jsou označena celými čísly od 1 do $N$ v pořadí od západu na východ, tzn. město $p$ leží východně od města $q$ právě když $p > q$. Žádná dvě města neleží na stejném poledníku. Mezi městy je organizována letecká přeprava pomocí přímých letů mezi dvojicemi měst (bez mezipřistání). Je znám seznam všech provozovaných přímých letů, všechny lety jsou obousměrné. Nacházíte se v nejzápadnějším městě (tj. městě číslo 1) a chcete podniknout následující cestu. Budete cestovat letecky stále směrem ze západu na východ, dokud nedoletíte do nejvýchodnějšího města (tj. města číslo $N$). Tam se otočíte a budete se letecky vracet zpět, pouze směrem z východu na západ, dokud nedoletíte do výchozího místa vaší cesty. Žádné město nesmí být navštíveno více než jednou, s výjimkou výchozího města, které navštívíte přesně dvakrát (na začátku a na konci vašeho výletu). Cestou nesmíte použít žádný jiný druh přepravy než zadané letecké spoje. Vaším úkolem je nalézt rozpis průběhu cesty, který umožní navštívit co nejvíce měst při splnění všech uvedených podmínek. Máte tedy vytvořit program, jehož vstupem bude počet měst $N$ (víte, že $N \leqq 100$) a seznam existujících přímých letů mezi městy (seznam dvojic měst letecky spojených). Výstupem programu má být údaj o počtu navštívených měst a jejich seznam v pořadí, v jakém je navštívíte, popř. sdělení, že úloha nemá řešení.

Úlohu si můžeme snadno převést do grafové podoby (viz kap. 9 o grafech). Města představují vrcholy neorientovaného neohodnoceného grafu a letecké spoje jeho hrany. Úkolem je nalézt v tomto grafu dvě různé cesty vedoucí z vrcholu 1 do vrcholu $N$ takové, že nemají žádný společný vrchol (kromě krajních vrcholů 1 a $N$), každá z nich prochází vrcholy grafu v rostoucím pořadí čísel a dohromady obsahují co největší počet vrcholů grafu.

K řešení úlohy nemůžeme použít žádný ze standardních grafových algoritmů pro hledání cest v grafu. Uvědomte si, jak složitě spolu souvisejí požadavky kladené na obě hledané cesty a kritérium pro určení nejlepšího výsledku. Není možné nalézt v grafu nejprve cestu z vrcholu 1 do vrcholu $N$ obsahující maximální počet vrcholů a pak k ní vyhledat zpáteční cestu, která neprochází žádným vrcholem první cesty a navštěvuje co

nejvíce ze zbývajících vrcholů. Takový postup by nemusel vést ke správnému výsledku, jak ukazuje následující příklad. Nechť $N = 6$ a existuje celkem 9 letů mezi dvojicemi měst 1–2, 1–3, 1–6, 2–4, 2–5, 3–4, 4–5, 4–6 a 5–6 (viz obr. 31). Cesta z vrcholu 1 do vrcholu 6 s maximálním počtem vrcholů je 1–2–4–5–6, zpět se pak ale dostaneme jedině přímým letem 6–1, takže celkově jsme navštívili pět měst. Kdybychom však cestou z 1 do 6 letěli po trase 1–2–5–6, zpět bychom se mohli vrátit cestou 6–4–3–1, takže dohromady bychom navštívili všech šest měst.




*Obr. 31* Příklad letů mezi 6 městy

Ukážeme si alespoň v náznaku, jak se úloha může řešit metodou dynamického programování. Pro jednoduchost se nejprve omezíme pouze na nalezení počtu měst navštívených během nejlepší cesty, tzn. bez určení konkrétních čísel navštívených měst a pořadí jejich návštěvy. Zavedeme si hodnoty $A_{I,J}$, které budou udávat maximální možný součet délek dvou disjunktních cest, z nichž jedna vede z vrcholu 1 do vrcholu $I$ a druhá z vrcholu 1 do vrcholu $J$. Délkou cesty zde rozumíme počet hran tvořících cestu (jinak to ani nejde, graf není ohodnocený, se skutečnými vzdálenostmi měst se v úloze vůbec nepracuje). Pojmem disjunktní cesty rozumíme cesty, které nemají žádný společný vrchol (s výjimkou výchozího vrcholu číslo 1). Z definice $A_{I,J}$ přímo plyne, že $A_{I,J} = A_{J,I}$, takže stačí počítat pouze hodnoty $A_{I,J}$ pro $J \leqq I$. Průběžně počítané hodnoty $A_{I,I}$ pro $I < N$ pro nás nemají žádnou cenu, důležitý však bude údaj $A_{N,N}$, který je hledaným výsledkem celé úlohy.

Hodnoty $A_{I,J}$ budeme počítat postupně podle rostoucího indexu $I$. Ukládat je budeme do připravené pomocné matice $A[1..N, 1..N]$, z níž však využijeme pouze část (trojúhelník pod hlavní diagonálou, tj. necelou



polovinu). Nejprve položíme z technických důvodů $A_{1,1} = 0$. Index $I$ se pak bude zvyšovat od 2 do $N$ a pro každé takové $I$ budeme volit $J$ z rozmezí od 1 do $I - 1$ a počítat hodnotu $A_{I,J}$. Na závěr navíc určíme stejným způsobem ještě výsledné $A_{N,N}$.

Zbývá ukázat, jakým způsobem spočítáme nějakou hodnotu $A_{I,J}$ pro $I > 1$. Podle výše uvedené definice $A_{I,J}$ odpovídá hodnotě $A_{I,J}$ jistá dvojice cest vedoucích z vrcholu 1 do vrcholů $I$ a $J$. Vrcholu $I$ musí na této cestě předcházet nějaký vrchol $K$ s číslem menším než $I$. Jistě je $K \neq J$ (vzhledem k disjunktnosti obou cest), může být ovšem $K > J$ nebo $K < J$. Počítaná hodnota $A_{I,J}$ je proto o 1 větší než nějaká hodnota $A_{K,J}$ nebo $A_{J,K}$ taková, že $1 \leqq K < I$, $K \neq J$, a v grafu existuje hrana z vrcholu $K$ do vrcholu $I$. Protože nám jde o získání maximální hodnoty $A_{I,J}$, stačí prozkoumat všechna taková $A_{K,J}$ (pro $K > J$) a $A_{J,K}$ (pro $K < J$), vzít z nich maximum a zvětšit ho o 1.

Tím máme vyřešenu úlohu s určením maximálního počtu navštívených měst. Ukážeme si ještě, jak lze vyhledat seznam měst navštívených během cesty (samozřejmě ve správném pořadí). Stejně jako v úloze o triangulaci mnohoúhelníku (kap. 14.2) si pomůžeme zavedením dalších hodnot $P_{I,J}$ ukládaných do druhého pomocného pole $P[1..N, 1..N]$. Hodnoty $P_{I,J}$ budou definovány pro tytéž dvojice indexů $I$, $J$ jako hodnoty $A_{I,J}$ a budou se s nimi také zároveň počítat. Hodnota $P_{I,J}$ udává číslo vrcholu, který je bezprostředním předchůdcem vrcholu $I$ na té cestě z 1 do $I$, která byla zvolena pro stanovení optimální hodnoty $A_{I,J}$. Je tedy rovna vybranému indexu $K$ z předchozího odstavce. Uložení hodnoty $P_{I,J}$ zároveň s výpočtem $A_{I,J}$ je triviální. Po skončení výpočtu a stanovení hodnot $A_{N,N}$ a $P_{N,N}$ můžeme poměrně snadno zrekonstruovat obě zvolené disjunktní cesty vedoucí z vrcholu 1 do vrcholu $N$, které vedly k získání maximálního $A_{N,N}$. Stačí nám k tomu pouze zaznamenané údaje $P_{I,J}$. Postupujeme od vrcholu $N$ zpět směrem k vrcholu 1 a souběžně vypisujeme obě cesty. Obecný krok tohoto zpětného průchodu vypadá následovně. Jsme-li momentálně u $P_{K,L}$ a platí $P_{K,L} = M$, je předchůdcem vrcholu $K$ na jedné z cest vrchol $M$. Pokud $M > L$, přejdeme na hodnotu $P_{M,L}$, která nám dá v dalším kroku předchůdce vrcholu $M$ (na téže cestě). Jestliže naopak $M < L$, přejdeme na hodnotu $P_{L,M}$, pomocí níž v dalším kroku získáme předchůdce vrcholu $L$ (na druhé z vytvářených cest).

Časová a paměťová složitost popsaného algoritmu je stejná jako u úloh v kap. 14.1 a 14.2. Opět pracujeme s pomocnou pamětí velikosti

$O(N^2)$ a postupně počítáme hodnoty jednotlivých jejích prvků. Spočítání každé z hodnot představuje práci $O(N)$, takže časová složitost algoritmu vychází $O(N^3)$. V případě výpisu nalezených cest má zpětný průchod polem $P$ jen lineární časovou složitost, neboť v každém kroku je jedna z cest prodloužena o jeden vrchol a obě cesty dohromady obsahují nejvýše všech $N$ vrcholů.

## CVIČENÍ

**1.** Upravte řešení úlohy o násobení matic z kap. 14.1 tak, aby výsledkem výpočtu byl nejen minimální potřebný počet násobení prvků matic, ale také informace o tom, v jakém pořadí se mají jednotlivé matice násobit, aby se tohoto minimálního počtu elementárních násobení dosáhlo.

**2.** Zamyslete se nad posledním řešením uvedeným v kap. 14.2. Má-li daný mnohoúhelník více různých triangulací minimální velikosti, která z nich se vytiskne?

**3.** Naprogramujte řešení úlohy o cestování uvedené v kap. 14.3.



# TECHNIKY NÁVRHU EFEKTIVNÍCH ALGORITMŮ

Již ve druhé kapitole jsme si řekli, že dobrý programátor při řešení úlohy nejprve posoudí různé možné postupy a vybírá z nich ten, který mu zdá nejvhodnější. Mezi důležitá hlediska, podle nichž vybíráme algoritmus řešení, patří časová a paměťová složitost výsledného programu. Budeme se snažit vytvořit program, pomocí něhož vyřešíme danou úlohu co nejrychleji a přitom s rozumnými paměťovými nároky.

Neexistuje žádný obecný návod, jakým způsobem vytvářet efektivní algoritmy. Problémy řešené na počítači jsou natolik rozmanité, že nějaký jednotný postup při jejich řešení není vůbec možný. Přesto se ale pokusíme ukázat alespoň některé základní obraty, které se při návrhu efektivních algoritmů opakovaně používají. Každé z těchto technik je věnována jedna z následujících podkapitol.

## 15.1 Odstranění opakovaných výpočtů

Naše první rada je velmi jednoduchá a s jejím uplatněním jsme se již vícekrát setkali v předcházejících kapitolách knihy. V programech se často stává, že s nějakou hodnotou je třeba vykonat několik různých výpočtů. Tato hodnota přitom není přímo součástí uložených vstupních dat, ale musíme si ji předem spočítat. Bylo by naprosto zbytečné počítat ji opakovaně vícekrát vždy, když ji potřebujeme. Vypočítáme ji pouze jednou a uložíme si ji do pomocné proměnné pro opakované využití. Celý výpočet se tím urychlí za cenu jistého paměťového prostoru navíc. Zkušení programátoři tento postup běžně užívají i v situacích, kdy nepřináší příliš zásadní zrychlení výpočtu. Někdy je však výsledný efekt opravdu významný. Pokud je opakovaný výpočet hodnoty časově náročný, jeho odstranění může zlepšit i asymptotickou časovou složitost algoritmu. V 9. kapitole věnované efektivitě rekurzivních algoritmů jsme si ukázali příklady, jak hodně lze tímto způsobem zvýšit rychlost výpočtu. Programy, které měly původně exponenciální časovou složitost, získaly po úpravě složitost lineární.

## 15.2 Přímé generování požadovaných údajů

Při řešení některých úloh potřebujeme nalézt všechny údaje jisté vlastnosti. Těmito údaji mohou být například celá čísla nebo $k$-tice čísel z předem známého rozmezí, prvních $N$ kladných celých čísel s danou vlastností apod. Úkol je možné řešit jednoduše tak, že procházíme postupně všechny údaje z daného rozmezí a každý otestujeme, zda vyhovuje požadavkům úlohy. Pokud by nebylo stanoveno rozmezí, ale třeba celkový počet $N$ hledaných kladných celých čísel, budeme postupně zkoumat všechna kladná celá čísla v rostoucím pořadí tak dlouho, až najdeme potřebných $N$ čísel splňujících zadané podmínky. Takovýto primitivní postup řešení bývá dosti neefektivní, zvláště je-li rozmezí zkoumaných údajů rozsáhlé a přitom obsahuje jen několik málo vyhovujících hodnot. Výpočet značně urychlíme, jestliže místo postupného testování všech údajů najdeme způsob, jak přímo generovat pouze hodnoty zadané vlastnosti. Pokud takový předpis neznáme, můžeme alespoň vhodným způsobem vybírat jenom ty údaje, u nichž je vůbec nějaká naděje, že budou splňovat podmínky úlohy. Místo celého přípustného rozmezí hodnot pak stačí testovat takto vybrané údaje. Při vhodně zvoleném výběru se počet potřebných testů značně sníží, což může vést až ke zlepšení asymptotické časové složitosti programu. Na dvou konkrétních příkladech si ukážeme, jakým způsobem se tato metoda používá a jaké zrychlení výpočtu může přinést.

### Příklad 1

Množina celých čísel M je definována následujícími pravidly:

a) číslo 1 je v M

b) jestliže číslo $x$ je z M, potom také $2x + 1$ a $3x + 1$ jsou z M

c) žádná jiná čísla než ta, která vzniknou pomocí předchozích dvou pravidel, do množiny M nepatří.

Prvními čísly množiny M jsou tedy 1, 3, 4, 7, 9, 10, 13, ... Sestavte algoritmus, který pro dané $N$ určí $N$ nejmenších čísel z množiny M.

*Řešení*

Jde o klasickou úlohu, kterou lze nalézt v mnoha publikacích o programování (např. [3], [4], [6], následně pak [18]). Jednoduché řešení může být založeno na postupném zkoumání všech celých čísel 1, 2, 3, ... atd., zda



požadovaného tvaru, a zda tedy patří do množiny M. Způsob testo- zda je zvolené číslo $x$ z množiny M, nás snadno napadne. Postupně střídavě odčítáme 1 a zkoušíme vzniklý rozdíl beze zbytku vydělit na nebo třemi tak dlouho, až se dostaneme k jedničce. Při tomto vání však musíme být velmi opatrní. Po odečtení jedničky můžeme t číslo, které je beze zbytku dělitelné jak dvěma, tak také třemi. om může být důležité, které z těchto dělení provedeme. Například testování čísla 13 získáme po odečtení 1 hodnotu 12, tu vydělíme na a dostaneme 6, odečteme 1 a obdržíme výsledek 5, který není telný dvěma ani třemi. Nemůžeme však ještě tvrdit, že 13 nepatří do Pokud totiž hodnotu 12 vydělíme třemi, získáme číslo 4, to zmenšené dává 3 a trojka je již jasně beze zbytku dělitelná třemi s výsledkem Číslo 13 tedy do množiny M patří. Správné otestování, zda číslo náleží M, vyžaduje systematicky vyzkoušet všechny možnosti dělení, což je mi pracné a pomalé.

Mnohem lepší je přímo generovat čísla náležející do množiny M ve stupném pořadí. Pro získání nového prvku množiny M vždy využijeme losti předcházejících čísel z M. Označíme-li prvky množiny M postup- v rostoucím pořadí symboly $m_1, m_2, m_3, \ldots$, pak pro nově počítaný rvek $m_i$ platí: $m_i$ je nejmenší číslo větší než $m_{i-1}$ takové, že ho lze yjádřit buď ve tvaru $2m_j + 1$ nebo $3m_k + 1$ pro nějaká $j$, $k$ menší než $i$. Můžeme si tedy postupně ukládat všechna již nalezená čísla z množiny M nové počítat vždy na základě této charakterizace.

Bylo by však naprosto zbytečné procházet pokaždé celým seznamem již známých prvků množiny M. Místo toho raději využijeme skutečnosti, že čísla z M získáváme a ukládáme v rostoucím pořadí. Zavedeme si dva pomocné indexy $j$, $k$, které nám budou v každém okamžiku určovat, že pro další prodloužení hledané posloupnosti je třeba posuzovat hodnoty $2m_j+1$ a $3m_k+1$. Pokud již máme čísla $m_1, m_2, m_3, \ldots, m_{i-1}$ a hledáme číslo $m_i$, indexy $j$, $k$ budou nastaveny tak, že platí:

$$2m_j + 1 > m_{i-1} \quad \wedge \quad 2m_{j-1} + 1 \leqq m_{i-1}$$
$$3m_k + 1 > m_{i-1} \quad \wedge \quad 3m_{k-1} + 1 \leqq m_{i-1}$$

Prvek $m_i$ nyní určíme velmi snadno, je jím menší z hodnot $2m_j+1, 3m_k+$ $+ 1$. Zároveň se stanovením nového prvku množiny M se musí posunout ten z indexů $j$, $k$, který k získání hodnoty $m_i$ přispěl. Pokud tedy $m_i =$ $= 2m_j + 1$, zvýšíme index $j$ o 1, podobně když $m_i = 3m_k + 1$, zvýšíme

o 1 index $k$. Nesmíme zapomenout, že může nastat také rovnost $2m_j + 1 = 3m_k + 1$. V takovém případě se musí posunout oba indexy $j$, $k$.

Získali jsme tak elegantní řešení, které má zjevně časovou složitost $O(N)$. Za zrychlení výpočtu jsme zaplatili zavedením pomocného pole velikosti $N$, do kterého si musíme ukládat všechny nalezené prvky množiny M.

```
program GenerovaniCisel;
{Nalezení prvních N čísel z množiny M definované pravidly:
 a) 1 je z M
 b) když x je z M, pak 2x+1 a 3x+1 jsou z M
 c) žádné jiné číslo nepatří do M
 Postup: přímé generování na základě znalosti předchozích}

const MaxN = 100;                        {max. možná hodnota N}
var M: array[1..MaxN] of integer;        {uložení prvků z M}
    N: integer;                          {počet hledaných čísel}
    I, J, K: integer;
    HI, HJ, HK: integer;

begin
write('Počet čísel: ');
readln(N);
M[1] := 1; write(1:5);                   {první číslo množiny z M}
J := 1; HJ := 3;
K := 1; HK := 4;
for I:=2 to N do
  begin
  if HJ < HK then HI := HJ else HI := HK;
  M[I] := HI;       {HI je hodnota v pořadí I-tého čísla z M}
  write(HI:5);            {vytiskneme I-té číslo z množiny M}
  if HI = HJ then
    begin
    J := J + 1;                          {posunutí indexu J}
    HJ := 2 * M[J] + 1
    end;
  if HI = HK then
    begin
    K := K + 1;                          {posunutí indexu K}
    HK := 3 * M[K] + 1
```



```
    end
nd;
iteln
d.
```

**íklad 2**

ozložte zadané kladné celé číslo $N$ na součet dvou třetích mocnin kladných celých čísel. Nalezněte všechna řešení.

*ešení*

Číslo $N$ chceme vyjádřit ve tvaru $N = A^3 + B^3$ pro vhodná kladná celá čísla $A$, $B$. Je zřejmé, že žádné z čísel $A$, $B$ nepřekročí rozmezí od 1 do $N$. Řešení úlohy je proto velmi snadné. Prozkoumáme všechny dvojice čísel z tohoto rozmezí a pro každou z nich zjistíme, zda splňuje stanovenou podmínku. Časová složitost tohoto jednoduchého postupu je $O(N^2)$, neboť zkoušíme $N$ možných hodnot proměnné $A$ a pro každou z nich $N$ hodnot proměnné $B$.

Výpočet můžeme urychlit vhodným snížením počtu zkoumaných dvojic čísel. Lehce zjistíme, že stačí uvažovat čísla $A$ z rozmezí od 1 do třetí odmocniny z $N$, vyšší hodnoty A po umocnění na třetí samy o sobě překračují $N$. Pro každé takové $A$ potom stačí zkoumat možné hodnoty proměnné $B$ od 1 do $A$, neboť se můžeme omezit na vyhledávání takových dvojic $A$, $B$, v nichž $A \geqq B$. Po této úpravě má algoritmus časovou složitost $O(N^{2/3})$.

Další zásadní snížení počtu dvojic $A$, $B$, jejichž otestování postačuje k získání výsledku, vyžaduje trochu hlubší úvahu. V předchozím řešení jsme zbytečně zkoumali všechny takové dvojice, ve kterých byla obě čísla „hodně malá“. Pro vyhovující dvojici čísel $A$, $B$, kde $A \geqq B$, přitom rozhodně musí platit, že je-li $B$ co nejmenší (tj. blízké 1), bude $A$ co největší (tj. blízké $N^{1/3}$). Naopak je-li $B$ co největší, je blízké $A$ a obě hodnoty nejsou příliš vzdáleny od $(N/2)^{1/3}$. Začneme proto sledovat pouze ty dvojice $A$, $B$, u nichž máme naději na nalezení řešení. Vyjdeme od prvního z uvedených „extrémů“ a zvolíme počáteční hodnoty proměnných tak, že $A$ je rovno celé části z $N^{1/3}$ a $B$ je rovno 1. Nyní určíme hodnotu výrazu $A^3 + B^3$. Pokud je větší než $N$, snížíme $A$ o 1 (hodnotu $B$ snížit už nemůžeme), pokud je menší než $N$, zvýšíme $B$ o 1

(hodnotu $A$ zvýšit nemůžeme). K nové dvojici $A$, $B$ opět spočítáme $A^3 + B^3$ a dále postupujeme stejným způsobem. Kdykoli je $A^3 + B^3$ příliš velké, hodnotu výrazu musíme snížit zmenšením $A$ o 1. Nemělo by žádnou cenu snižovat hodnotu proměnné $B$, neboť všechny nižší hodnoty $B$ jsme již dříve prozkoumali. Nízkou hodnotu výrazu $A^3+B^3$ opravíme obdobně zvýšením $B$ o 1. Pokud narazíme na dvojici čísel splňující podmínku $A^3 + B^3 = N$, našli jsme řešení úlohy. Jelikož máme nalézt všechna řešení, upravíme obě proměnné $A$, $B$ ($A$ snížíme, $B$ zvýšíme) a pokračujeme ve výpočtu stejným způsobem. Takto se nám obě hodnoty proměnných $A$, $B$ k sobě postupně přibližují a procházejí všemi nadějnými dvojicemi čísel. Výpočet končí ve chvíli, kdy se setkají, neboť stačí zkoumat $A$, $B$ taková, že $A \geq B$. V každém kroku výpočtu se buď sníží $A$ o 1, nebo zvýší $B$ o 1 (popř. obojí), takže nejvýše po zhruba $N^{1/3}$ krocích výpočet skončí. Je tedy jistě konečný a má časovou složitost $O(N^{1/3})$. Řádového zvýšení efektivity proti předchozím řešením jsme dosáhli vhodným omezením počtu dvojic zkoumaných čísel.

Podobné řešení této úlohy můžete nalézt také v knize [14], kde je navíc trochu precizněji zdůvodněna jeho správnost. Zde si ukážeme jiný, snad trochu názornější způsob naprogramování, než jaký je uveden v [14].

```
program RozkladNaTretiMocniny;
{Rozklad daného kladného celého čísla na součet dvou třetích
 mocnin kladných celých čísel}
var N: integer;        {rozkládané číslo}
    A, B: integer;     {rozklad N = A^3 + B^3}
    A3, B3: integer;   {hodnoty A^3, B^3}
begin
write('Rozkládané číslo: ');
readln(N);
A := 1;                {výpočet iniciální hodnoty A}
while A * A * A < N do A := A + 1;
A := A - 1;
A3 := A * A * A;
B := 1;                {iniciální hodnota B}
B3 := 1;
while A >= B do
  if A3 + B3 > N then
    begin A := A-1; A3 := A * A * A end
  else if A3 + B3 < N then
```



```
    begin B := B+1; B3 := B * B * B end
  else  {A3 + B3 = N {...} máme řešení}
    begin
    writeln(A:8, B:8);
    A := A-1; A3 := A * A * A;
    B := B+1; B3 := B * B * B
    end
end.
```

Někdo by mohl namítnout, že algoritmus se stejnou časovou složitostí $O(N^{1/3})$ může být mnohem jednodušší. Stačí zkoumat všechna celá čísla $A$ z rozmezí od 1 do $N^{1/3}$ a pro každé takové $A$ přímo spočítat jedinou možnou hodnotu $B$ výrazem $(N - A^3)^{1/3}$. Pokud má tento výraz celočíselnou hodnotu, je roven hledanému $B$ a máme řešení úlohy. Jestliže není celočíselný, pak pro toto $A$ žádné vyhovující $B$ neexistuje a musíme zkusit další $A$. Tento postup je samozřejmě také správný a má asymptotickou časovou složitost $O(N^{1/3})$. Naše předcházející ukázkové řešení úlohy je ale přesto lepší. Všechny výpočty se v něm totiž odehrávají pouze pomocí celočíselných proměnných a jednoduchých operací s nimi. U tohoto posledního postupu je nutné počítat v každém kroku třetí odmocninu. Přitom se nemůžeme vyhnout práci s reálnými proměnnými a při programování navíc ještě použití nějakých standardních funkcí. Celý výpočet proto bude značně pomalejší.

### 15.3 Výpočet nové hodnoty na základě předchozí

V některých úlohách je zapotřebí počítat postupně posloupnost nějakých hodnot a tyto hodnoty pak dále zpracovávat. Výpočet každé z hodnot představuje jisté netriviální množství práce, takže celková časová náročnost výpočtu je rovna součinu počtu zjišťovaných hodnot a počtu operací potřebných k získání jedné hodnoty. Často se ovšem stává, že po sobě jdoucí hodnoty spolu nějakým způsobem souvisejí. Potom se mnohdy vyplatí nepočítat novou hodnotu samostatně od začátku, ale určit ji na základě již známé předchozí hodnoty, popř. pomocí více hodnot. Přitom je možné využívat také další pomocné údaje a mezivýsledky, které pocházejí z výpočtu předcházejících hodnot a které jsme si za tímto účelem zaznamenali. Takovýto způsob stanovení nového členu posloupnosti může být řádově rychlejší, čímž se sníží i časová složitost celého algoritmu.

Za zrychlení někdy musíme zaplatit zavedením pomocných proměnných navíc, v nichž se budou průběžně udržovat údaje potřebné k výpočtu nové hodnoty na základě předchozích znalostí. Praktické použití tohoto postupu si opět ukážeme na příkladech.

### Příklad 1

Máme dánu posloupnost $N$ celých čísel. Nalezněte v ní úsek po sobě jdoucích čísel s co největším součtem prvků. Řešením úlohy je tento maximální součet.

*Řešení*

Různé varianty řešení můžete nalézt například v [2] nebo [18]. Nejjednodušší řešení úlohy je naprosto přímočaré. Posloupnost čísel si nejdříve načteme a uložíme do pole. Projdeme všechny souvislé úseky v poli, tj. úseky od $i$-tého po $j$-tý prvek pro všechny dvojice indexů $i, j$ takové, že $1 \leqq i \leqq j \leqq N$. Spočítáme součet prvků v každém úseku a z těchto součtů vybereme maximum. Všechny součty úseků se v tomto řešení počítají zcela nezávisle na sobě, což vede k časové složitosti algoritmu $O(N^3)$.

Lepší řešení využívá skutečnost, že součet čísel v úseku od $i$-tého po $j$-tý prvek můžeme snadno spočítat, známe-li již součet čísel v předchozím úseku od $i$-tého po $(j-1)$-ní prvek. Součty úseků se stejným začátkem tedy nebudeme počítat nezávisle na sobě, ale postupně s rostoucím indexem jejich koncového prvku budeme vždy součet počítat na základě předchozího. Touto úpravou získáme časovou složitost $O(N^2)$.

Úlohu je ale možné vyřešit ještě lépe. Nejlepší algoritmus má lineární časovou složitost $O(N)$ a postačí mu jednou projít danou posloupností čísel od začátku do konce. Při tomto průchodu si budeme ve dvou proměnných udržovat maximální dosud nalezený součet souvislého úseku a dosavadní průběžný součet právě sledovaného úseku. Po přečtení každého z čísel posloupnosti obě tyto hodnoty zaktualizujeme. Nejprve otestujeme, zda je výhodnější přidat nové číslo ke sledovanému úseku nebo zda je lepší na „minulost" zapomenout a novým číslem zahájit další úsek. Výhodnější je ta alternativa, která nám dá vyšší průběžný součet úseku. Nastavíme tedy nový průběžný součet podle zvolené příznivější alternativy. Potom ještě tento součet porovnáme s dosud maximálním nalezeným součtem, a je-li vyšší, zaznamenáme také nové nalezené maximum.



Řešení úlohy si předvedeme také ve tvaru programu. Program očekává na vstupu posloupnost celých čísel. Poslední z čísel musí být bezprostředně následováno příznakem konce vstupního souboru. Čtená čísla jsou ihned průběžně zpracovávána, takže nepotřebujeme žádné pole pro jejich uložení. Díky tomu není ani nijak omezen jejich počet.

```
program MaxSoucet;
{V dané posloupnosti celých čísel hledá maximální součet
 souvislého úseku.}

var Cislo: integer;          {čtené číslo}
    Max: integer;            {celkový maximální součet úseku}
    Soucet: integer;         {průběžný součet}

begin
writeln('Zadejte posloupnost čísel:');
read(Cislo);                 {první číslo}
Max := Cislo;
Soucet := Cislo;             {zatím máme jednoprvkový úsek}
while not eof do
  begin
  read(Cislo);               {nový člen posloupnosti}
  if Soucet < 0 then
    Soucet := Cislo          {výhodnější je začít nový úsek}
  else
    Soucet := Soucet+Cislo;  {prodloužení úseku}
  if Soucet > Max then
    Max := Soucet            {nový maximální součet}
  end;
writeln('Maximální součet souvislého úseku: ', Max)
end.
```

### Příklad 2

Je dána posloupnost celých čísel. Její délka není předem známa, posloupnost může být velmi dlouhá. Určete v ní délku co nejdelšího souvislého úseku, v němž se libovolná dvě čísla liší maximálně o 1. Např. pro posloupnost 5 7 6 7 7 8 8 7 9 9 9 bude výsledkem 5 (což je délka úseku 7 7 8 8 7).

*Řešení*

Úloha byla použita jako soutěžní úloha domácího kola 43. ročníku MO kategorie P. Souvislý úsek posloupnosti, který je tvořen čísly lišícími se nejvýše o 1, tam byl nazýván hladkým úsekem. Toto přirozené krátké označení budeme používat i zde. Hladký úsek je tedy takový souvislý úsek, který je buď tvořen pouze opakujícími se stejnými čísly, nebo obsahuje dvě hodnoty lišící se o 1, každou v libovolném počtu a v jakémkoliv pořadí.

Podle zadání neznáme délku zkoumané posloupnosti, takže si ji nemůžeme předem přečíst a uložit do pole, kde bychom pak hladké úseky vyhledávali. Nemůžeme si ji ani načítat po jednotlivých hladkých úsecích, neboť samozřejmě neznáme ani žádné horní omezení na délku maximálního hladkého úseku. Výsledek tedy musíme získat v průběhu jednoho (nebo popř. několikrát opakovaného) postupného čtení zadané posloupnosti čísel. V hrubých rysech je postup výpočtu jasný: v posloupnosti postupně vyhledáme všechny hladké úseky, určíme délku každého z nich a z těchto délek vezmeme maximum. Problémy nejsou ani s určením délky právě sledovaného hladkého úseku. V několika málo proměnných si budeme během čtení posloupnosti udržovat informace, jakými hodnotami je sledovaný úsek tvořen a jakou již má délku. Konec úseku nám určí „cizí" hodnota, popř. konec celé posloupnosti.

Jediná potíž spočívá v tom, že dva po sobě jdoucí hladké úseky v posloupnosti se mohou částečně překrývat. Pokud bychom začali další hladký úsek zkoumat až v okamžiku ukončení předchozího, nemuseli bychom správné řešení úlohy vůbec nalézt! Například v posloupnosti ze zadání úlohy bychom postupně určili hladké úseky 5, 7 6 7 7, 8 8 7 a 9 9 9. Nejdelší z nich má délku 4, což je chybný výsledek.

První nápad, jak tuto potíž odstranit, nás povede k dosti pomalému a nešikovnému řešení. Když nemůžeme začít zkoumat další hladký úsek až po ukončení předchozího, budeme předpokládat jeho začátek vždy o jedinou pozici v posloupnosti dále, než začínal předcházející úsek. Takto nemůžeme správné řešení minout. Musíme však opakovaně číst danou posloupnost od začátku, neboť jinak se k již přečteným hodnotám nedokážeme vrátit. To ve svém důsledku vede k řešení s časovou složitostí $O(N^2)$, kde $N$ je délka dané posloupnosti.

Lepšího řešení dosáhneme, jestliže budeme vždy při zkoumání jednoho hladkého úseku myslet již trochu napřed na to, že nás pak bude



zkat další hladký úsek. Budeme si navíc průběžně zaznamenávat, kolika stejnými hodnotami a jakými končí sledovaný hladký úsek. Díky tomu dokážeme celou úlohu vyřešit při jediném průchodu posloupností algoritmem s lineární časovou složitostí. Kdykoli skončí jeden hladký úsek posloupnosti a budeme začínat zpracovávat nový, dokážeme pomocí zaznamenaných údajů snadno určit, zda se nový úsek částečně překrývá s předchozím a kolik prvků toto překrytí činí. K překrytí sousedních hladkých úseků totiž dojde právě tehdy, jestliže se nově přečtené číslo liší o 1 od posledního čísla právě ukončeného hladkého úseku. Na základě znalosti, kolika stejnými čísly poslední hladký úsek končí, pak již snadno zahrneme tento počet do délky nového hladkého úseku. V posloupnosti se tudíž nemusíme vůbec vracet, budeme ji číst stále dál. V případě, že k překrytí úseků došlo, však nebudeme nový hladký úsek počítat „od nuly", ale na základě zaznamenaných informací budeme počítat, jako by již byl začátek nového úseku načten.

Řešení si ukážeme naprogramované v Pascalu.

```
program MaxHladkyUsek;
{V dané posloupnosti čísel určí délku maximálního souvislého
 úseku, jehož každá dvě čísla se liší
 nejvýše o 1 --- tzv.  hladkého úseku }

var Cislo:integer;          {právě přečtené číslo}
    Predchozi:integer;      {předchozí číslo}
    Index:integer;          {pořadové číslo přečteného čísla}
    Hladky:integer;         {index začátku sledovaného
                             hladkého úseku}
    Konstantni:integer;     {index začátku posledního
                             konstantního úseku}
    H1:integer;             {menší z hodnot v aktuálním úseku}
    H2:integer;             {větší z hodnot v aktuálním úseku}
    Max:integer;            {délka maximálního hladkého úseku}

begin
writeln('Zadejte posloupnost čísel: ');
read(Cislo);                {přečteno první číslo}
Index := 1;                 {jeho index je 1}
Hladky := 1;                {prvním číslem začíná hladký úsek}
Konstantni := 1;            {začíná také konstantní úsek}
```

```
H1 := Cislo;
Max := 1;

while not eof do
  begin
  Predchozi := Cislo;
  read(Cislo);                    {další člen posloupnosti}
  Index := Index+1;
  if Hladky = Konstantni then
    begin     {hladký úsek je zatím tvořen jedinou hodnotou}
    if Cislo = H1+1 then          {prodloužení hladkého úseku}
      begin
      H2 := Cislo;
      Konstantni := Index
      end
    else if Cislo = H1-1 then     {prodloužení hladkého úseku}
      begin
      H2 := H1;
      H1 := Cislo;
      Konstantni := Index
      end
    else if Cislo <> H1 then      {konec hladkého úseku, začíná
                                   nový úsek bez překrytí}
      begin
      if Index-Hladky > Max then
        Max := Index-Hladky;      {nová max. délka hlad. úseku}
      Hladky := Index;
      Konstantni := Index;
      H1 := Cislo
      end
    end
  else  {H2 = H1+1}
    begin      {sledovaný hladký úsek je tvořen 2 hodnotami}
    if (Cislo = H1) or (Cislo = H2) then
      begin                       {prodloužení hladkého úseku}
      if Cislo <> Predchozi then Konstantni := Index
      end
    else                          {konec hladkého úseku}
      begin
      if Index-Hladky > Max then
        Max := Index-Hladky;      {nová max. délka hlad. úseku}
```



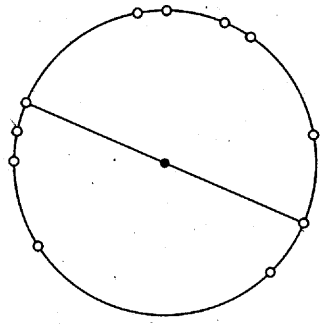
```
    if (Cislo = H2+1) and (Predchozi = H2) then
      begin           {nový úsek se překrývá s předchozím}
      H1 := H2;
      H2 := Cislo;
      Hladky := Konstantni
      end
    else if (Cislo = H1-1) and (Predchozi = H1) then
      begin           {nový úsek se překrývá s předchozím}
      H2 := H1;
      H1 := Cislo;
      Hladky := Konstantni
      end
    else              {začíná nový úsek bez překrytí}
      begin
      H1 := Cislo;
      Hladky := Index
      end;
    Konstantni := Index
    end
  end
end;

if Index-Hladky+1 > Max then
  Max := Index-Hladky+1;          {nová max. délka hlad. úseku}
writeln('Délka maximálního hladkého úseku: ',Max)
```

### Příklad 3

Na kružnici je rozmístěno $N$ bodů tak, že všechny jejich vzájemné vzdálenosti měřené po obvodu kružnice jsou celočíselné. Jsou dány vzájemné vzdálenosti všech dvojic bodů, které spolu na obvodu kružnice sousedí. Zjistěte, zda mezi danými $N$ body existují dva takové, jejichž spojnice prochází středem kružnice. Pokud ano, jednu takovou dvojici bodů nalezněte.

*Řešení*

Všechny vzájemné vzdálenosti daných bodů uvažujeme v této úloze tak, že jsou měřeny po obvodu kružnice v kladném smyslu orientace. Nadále budeme pro jednoduchost hovořit již jen o vzdálenosti bodů.



*Obr. 32* Body na kružnici s vyznačeným řešením úlohy

Vzájemná vzdálenost $V$ hledané dvojice bodů je rovna přesně polovině délky kružnice. Délku kružnice snadno určíme jako součet vzdáleností všech dvojic sousedních bodů, hodnotu $V$ pak získáme pouhým vydělením dvěma. Pokud by nám vyšla délka kružnice lichá, úloha pochopitelně nemůže mít žádné řešení. Jinak potřebujeme nalézt dva takové body, které jsou od sebe vzdáleny přesně $V$.

Vstupní data, tj. vzájemné vzdálenosti dvojic sousedních bodů, si uložíme do pole celých čísel $D[1..N]$. Hodnota $D[i]$ pro $i < N$ udává vzdálenost bodů $A_iA_{i+1}$, poslední prvek $D[N]$ představuje vzdálenost $A_NA_1$. Polem $D$ budeme opakovaně procházet a ke každému bodu $A_j$, $j = 1, 2, \ldots, N-1$ budeme zkoušet najít takový bod $A_k$, aby vzdálenost $A_jA_k$ byla rovna $V$. Pro každé $j$ budeme počítat součet

$$D[j] + D[j+1] + D[j+2] + \ldots$$

tak dlouho, dokud nedosáhne hodnoty $V$ nebo ji nepřekročí. Jestliže se nám podaří po $m$ sčítáních získat přesný součet

$$D[j] + D[j+1] + \ldots + D[j+m] = V,$$

pak dvojice bodů $A_j$, $A_{j+m+1}$ je hledaným řešením. Pokud přesné hodnoty $V$ nedosáhneme, bod $A_j$ nevyhovuje a nebude součástí řešení. Tento postup je naprosto správný, je však zbytečně pomalý. Algoritmus má kvadratickou časovou složitost, neboť postupně testuje až $N$ výchozích bodů (parametr $j$) a pro každý z nich provádíme až $N$ sčítání (parametr $m$).



Zrychlení výpočtu je založeno na myšlence, že po zvýšení hodnoty $j$ nemusíme počítat celý součet

$$D[j] + D[j+1] + D[j+2] + \ldots$$

znovu od začátku, ale že můžeme s výhodou využít zaznamenanou předchozí hodnotu součtu. Ukážeme si to na konkrétním příkladu. Začali jsme počítat součet pro bod $A_1$ a zjistili jsme, že

$$D[1] + D[2] + D[3] + D[4] + D[5] < V,$$
$$D[1] + D[2] + D[3] + D[4] + D[5] + D[6] > V.$$

V tomto okamžiku víme, že bod $A_1$ nebude součástí řešení, neboť bod $A_6$ je k němu moc blízko a bod $A_7$ je už zase moc daleko. Zkusíme tedy vyšetřovat bod $A_2$. Proč bychom ale počítali znovu pracně celý součet

$$D[2] + D[3] + D[4] + \ldots,$$

když známe z předchozího kroku hodnotu

$$D[1] + D[2] + D[3] + D[4] + D[5] + D[6].$$

Stačí od ní odečíst $D[1]$ a rázem získáme součet

$$D[2] + D[3] + D[4] + D[5] + D[6].$$

Nemusíme mít žádné obavy, že bychom snad takto mohli neoprávněně přeskočit hledanou hodnotu $V$. Víme totiž, že

$$D[1] + D[2] + D[3] + D[4] + D[5] < V,$$

takže tím spíše také

$$D[2] + D[3] + D[4] + D[5] < V$$

a prvním „zajímavým“ součtem od bodu $A_2$ je skutečně náš lehce získaný

$$D[2] + D[3] + D[4] + D[5] + D[6].$$

Ten nyní porovnáme s hodnotou $V$ a podle potřeby ho ještě prodloužíme.

Upravený algoritmus nyní již snadno popíšeme a naprogramujeme. Od předchozího řešení se liší pouze tím, že vždy po nové volbě počátečního bodu $A_j$ (tj. po zvýšení hodnoty $j$ o 1) nepočítáme od začátku součet $D[j] + D[j+1] + \ldots$, ale využijeme minulou hodnotu tohoto součtu, od které odečteme $D[j-1]$. Celý postup si můžeme názorně představit tak, že po bodech na kružnici se pohybují dvě ukazovátka $J$ a $K$ vymezující začátek a konec zkoumaného úseku $A_J A_K$. Je-li úsek kratší než $V$, posune se ukazatel $K$ do dalšího bodu, a tím se úsek prodlouží. Pokud je naopak úsek $A_J A_K$ moc dlouhý, posune se $J$ a zkoumaný úsek se tím zkrátí. Tento proces se opakuje tak dlouho, dokud nedosáhneme úseku velikosti přesně $V$ nebo dokud ukazatel $K$ nepřekročí poslední bod $A_N$ (v takovém případě řešení neexistuje).

Algoritmus má lineární časovou složitost. Výpočet začíná ve stavu $J = 1$, $K = 1$, v každém kroku se zvýší buď $J$, nebo $K$, přičemž stále je $J \leqq K$. Výpočet končí nejpozději ve chvíli, když $K = N + 1$. Celkem se tedy provede méně než $2N$ kroků.

Program očekává na vstupu posloupnost celých čísel, která představují vzájemné vzdálenosti dvojic sousedních bodů v pořadí $A_1A_2$, $A_2A_3$, ..., $A_NA_1$. Pro zjednodušení zápisu opět předpokládáme, že hned za posledním číslem je na vstupu zapsán znak konce souboru.

```
program BodyNaKruznici;
{Nalezení dvojice bodů na průměru kružnice}
const MaxN = 100;          {maximální počet bodů na kružnici}
var D: array[1..MaxN] of integer;
                           {vzdálenosti sousedních bodů}
    N: integer;            {počet bodů}
    V: integer;            {délka půlkružnice}
    J, K: integer;         {vymezení zkoumaného úseku}
    Soucet: integer;       {délka zkoumaného úseku}

begin
{Zadání vstupních hodnot a výpočet délky půlkružnice:}
writeln('Vzájemné vzdálenosti dvojic sousedních bodů');
writeln('v pořadí A1A2, A2A3, {...}, ANA1:');
N := 0;
V := 0;
while not eof do
```



```
begin
N := N+1;
read(D[N]);
V := V+D[N]
end;
if odd(V) then       {lichá délka kružnice}
writeln('Lichá délka kružnice - úloha nemá řešení!')
else                 {sudá délka kružnice}
begin
V := V div 2;        {délka půlkružnice}
{Vlastní hledání dvojice bodů na průměru kružnice:}
J := 1; K := 1;
Soucet := 0;
while (Soucet <> V) and (K <= N) do
  if Soucet < V then
    begin
    Soucet := Soucet + D[K];
    K := K+1         {zvýšení horní meze zkoumaného úseku}
    end
  else
    begin
    Soucet := Soucet - D[J];
    J := J+1         {zvýšení dolní meze zkoumaného úseku}
    end;
if Soucet = V then
  writeln('Na průměru leží body s indexy ', J, ' a ', K)
else
  writeln('Úloha nemá řešení!')
end
end.
```

Všimněte si výrazné podobnosti právě uvedeného programu s programem uvedeným v př. 2, kap. 15.2 (rozklad daného čísla na součet dvou třetích mocnin). V obou případech jde o použití stejné techniky dvou střídavě se pohybujících indexů.

## 15.4 Předzpracování vstupních dat

Ke zrychlení programu nám někdy pomůže provedení vhodného předvýpočtu. Místo abychom z uložených vstupních dat přímo počítali výsledky algoritmem s časovou složitostí $f(N)$, rozdělíme celý výpočet do dvou (popř. více) po sobě jdoucích etap. V první etapě si vstupní data předzpracujeme a takto vzniklé mezivýsledky si uložíme. Druhá etapa je pak řešením vlastní úlohy na základě zaznamenaných výsledků první etapy výpočtu. Šikovné rozdělení práce do etap a volba vhodného tvaru ukládaných mezivýsledků může způsobit, že obě etapy řešení mají časovou složitost menší než původní $f(N)$. Vzhledem k tomu, že při výpočtu etapy následují po sobě, bude i časová složitost celého algoritmu ostře menší než $f(N)$. Zlepšení časové efektivity jsme dosáhli opět za cenu zvýšení paměťových nároků. Oproti přímočarému řešení potřebujeme navíc paměťový prostor pro uložení mezivýsledků, jehož velikost bývá často srovnatelná s rozsahem zpracovávaných vstupních dat.

### Příklad 1

Je dáno $N$-prvkové pole celých čísel. Určete číslo, které se v daném poli nachází nejvícekrát. Je-li takových čísel více (se stejným maximálním počtem výskytů), výsledkem úlohy bude jedno libovolné z nich.

*Řešení*

Náš první příklad je zcela triviální, ale velmi názorný. Jednoduché přímé řešení úlohy spočívá v tom, že postupně pro každý prvek pole projdeme celým polem a spočítáme, kolik shodných hodnot se v poli nachází. Průběžně si evidujeme maximální nalezený počet stejných čísel a prvek, pro který jsme tohoto maxima dosáhli. Časová složitost algoritmu je $O(N^2)$, neboť $N$-krát procházíme $N$-prvkovým polem.

Řešení úlohy zrychlíme, jestliže použijeme ten nejjednodušší a nejčastější předvýpočet, totiž setřídění dat. V první etapě výpočtu čísla v poli setřídíme podle velikosti. K tomu použijeme některý z běžných třídicích algoritmů s časovou složitostí $O(N \log N)$ (viz kap. 11.1). Setříděním se dostaly k sobě stejné hodnoty uložené v poli. Druhá etapa výpočtu, tj. vlastní nalezení výsledku, je pak již snadná. Stačí jednou projít polem a počítat v něm souvislé úseky stejných čísel. Tato fáze má časovou složitost $O(N)$. Složitost celého řešení je rovna součtu složitostí obou po sobě následujících etap a vychází $O(N \log N)$. Je tedy nižší, než byla časová složitost původního řešení bez předvýpočtu.



### Příklad 2

Je dána obdélníková matice celých čísel velikosti $N \times M$. Každý prvek matice má hodnotu 0 nebo 1. Nalezněte v ní co největší podmatici (tj. souvislý obdélníkový výřez) tvořenou samými jedničkami. Řešením úlohy je počet prvků této nalezené maximální jedničkové podmatice.

*Řešení*

Tato úloha je krásnou ukázkou toho, jak nám může ke zrychlení výpočtu pomoci i trochu komplikovanější předvýpočet. Její jednoduché řešení představuje postupné zkoumání všech podmatic, zda jsou tvořeny samými jedničkami. Každá podmatice je vymezena volbou svého levého, pravého, horního a dolního okraje, počet všech podmatic v matici $N \times M$ je proto řádově $N^2M^2$. Otestování všech prvků zvolené podmatice vyžaduje projít celou podmaticí, což znamená až $NM$ operací. Celkově má tedy tento triviální postup řešení časovou složitost $O(N^3M^3)$.

Opakovanému testování jednotlivých prvků matice se vyhneme tím, že si před vlastním vyhledáváním maximální jedničkové podmatice vhodně předzpracujeme vstupní data. Ke každé jedničce v dané matici si spočítáme, kolik dalších jedniček je v matici v souvislé řadě pod ní. O tento počet pak jedničku zvýšíme. Získané údaje si můžeme uložit přímo do původní matice, takže ani nebudeme potřebovat nějakou další datovou strukturu. Nulové prvky zůstanou zachovány a místo jedniček se v matici objeví kladná celá čísla. V případě potřeby by bylo možné po ukončení výpočtu snadno obnovit zadanou matici do původního tvaru. Na jednom konkrétním příkladě si nyní ukážeme, jak bude vypadat výsledek předvýpočtu:

$$\text{z matice } \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \text{získáme} \quad \begin{pmatrix} 0 & 3 & 1 & 0 & 5 \\ 3 & 2 & 0 & 4 & 4 \\ 2 & 1 & 3 & 3 & 3 \\ 1 & 0 & 2 & 2 & 2 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Časová náročnost uvedeného předvýpočtu je pouze $O(NM)$. Stačí jednou projít každým sloupcem matice ve směru zdola nahoru, nulové

prvky ponechat beze změny a jedničky zvýšit o hodnotu prvku ležícího bezprostředně pod ní.

Při vlastním hledání maximální jedničkové podmatice budeme již pracovat s upravenou maticí. Výsledná podmatice má jistě levý horní roh v některém nenulovém prvku matice. Vyzkoušíme tedy postupně všechny možné polohy levého horního rohu (řádově $NM$ možností). Pro každou volbu levého horního rohu podmatice budeme zkoumat, kde všude by tato podmatice mohla mít pravý horní roh. Pravý horní roh může buď splývat s levým horním rohem, nebo může ležet ve stejné řadě vpravo od něj. Postupovat vpravo můžeme tak dlouho, dokud nenarazíme na první nulu nebo na okraj matice (prozkoumáme tedy nejvýše $M$ možností). Nyní se teprve ukáže výhodnost provedeného předvýpočtu. Při postupném sledování možných poloh pravého horního rohu můžeme zároveň snadno počítat, jak velká je co největší jedničková podmatice se zvoleným umístěním obou horních rohů. Čísla na řádku mezi levým a pravým horním rohem nám určují hloubku sloupců jedniček směrem dolů, takže stačí vybrat z nich minimum. Velikost jedničkové podmatice se pak spočítá jako součin tohoto minima a vzdálenosti levého a pravého horního rohu. Minima i součiny se počítají průběžně při každém posunutí pravého horního rohu a představují konstantní počet provedených operací, neboť nové minimum se počítá vždy na základě předchozího. Tyto výpočty proto nijak nezatíží asymptotickou časovou složitost celého algoritmu. Ta je dána pouze počtem zkoumaných umístění levého a pravého horního rohu podmatice a je rovna $O(NM^2)$.

Řešení úlohy se tedy skládá ze dvou po sobě jdoucích etap. První je předvýpočet s časovou složitostí $O(NM)$, druhou etapou je vlastní vyhledání maximální jedničkové podmatice s časovou složitostí $O(NM^2)$. Proti původnímu primitivnímu řešení jsme tedy dosáhli velmi výrazného zrychlení výpočtu.

Popsané řešení nyní naprogramujeme v Pascalu. Program předpokládá, že žádný z rozměrů matice nepřekročí zvolenou konstantu $Max$ (pro jednoduchost není tato podmínka testována). Na vstupu očekává nejprve rozměry matice $N$, $M$ a pak obsah zkoumané matice po řádcích. Jako výsledek vytiskne velikost největší podmatice tvořené samými jedničkami. Nebylo by již žádným problémem doplnit do programu průběžnou evidenci, kde byla nalezena dosud největší jedničková podmatice. Program



by pak mohl vedle velikosti udávat také polohu maximální podmatice ze samých jedniček.

```
program JednickovaPodmatice;
{Určení velikosti maximální podmatice ze samých jedniček
 v zadané matici nul a jedniček}

const Max = 20;          {maximální velikost zkoumané matice}
var A: array[1..Max,1..Max] of integer;    {zkoumaná matice}
    N, M: integer;                         {velikost matice}
    I, J: integer;     {poloha levého horního rohu}
    K: integer;        {sloupec pravého horního rohu}
    Min: integer;      {minimum z délek sloupců jedniček}
    Vel: integer;      {velikost pravé zkoumané podmatice}
    MaxVel: integer;   {maximální velikost podmatice z 1}
    Dalsi: boolean;    {zkoumat další pravý horní roh?}

begin
{Vstup dat:}
write('Rozměry matice: ');
readln(N, M);
writeln('Obsah matice po řádcích:');
for I:=1 to N do
  for J:=1 to M do read(A[I,J]);

{Předvýpočet:}
for J:=1 to M do
  for I:=N-1 downto 1 do
    if A[I,J] = 1 then A[I,J] := A[I,J] + A[I+1,J];

{Vlastní výpočet:}
MaxVel := 0;
for I:=1 to N do
  for J:=1 to M do
    if A[I,J] > 0 then     {levý horní roh podmatice A[I,J]}
      begin
      K := J;              {pravý horní roh podmatice A[I,K]}
      Min := A[I,J];
      Vel := Min;          {šířka podmatice = 1, výška = Min}
      if Vel > MaxVel then
        MaxVel := Vel;      {našli jsme větší podmatici z 1}
```

```
    if K = M then
      Dalsi := false                    {není kam pokračovat}
    else
      begin
      K := K+1;
      Dalsi := A[I,K] > 0
      end;
    while Dalsi do {zkoumání dalších pravých horních rohů}
      begin
      if A[I,K] < Min then Min := A[I,K];
      Vel := Min * (K-J+1);        {velikost nové podmatice}
      if Vel > MaxVel then MaxVel := Vel;
      if K = M then
        Dalsi := false                  {není kam pokračovat}
      else
        begin
        K := K+1;
        Dalsi := A[I,K] > 0
        end
      end
  end;

{Výsledek:}
write('Velikost maximální podmatice ze samých jedniček: ');
writeln(MaxVel)
end.
```

Uvedený algoritmus je možné ještě dále vylepšovat. To podstatné, o co nám nyní šlo, tj. využití předvýpočtu ke snížení časové složitosti, jsme si však již plně předvedli na tomto řešení úlohy. Jednoduché zlepšení získáme na základě úvahy, že stejný předvýpočet jsme mohli provést v řádcích matice místo ve sloupcích. Tím bychom úlohu vyřešili v čase $O(N^2M)$ namísto $O(NM^2)$. Pokud by se rozměry $N$, $M$ hodně lišily, vyplatí se nám otestovat, který z nich je menší, a podle toho zvolit orientaci předvýpočtu. Pro $N > M$ je lepší použít výše popsané řešení s předvýpočtem ve sloupcích, naopak pro $N < M$ bude výhodnější počítat jedničky v řádcích.

Existuje ještě lepší řešení úlohy, které je založeno na předběžném stanovení počtů jedniček vždy nad i pod každým jedničkovým prvkem zkoumané matice. Myšlenkově je poněkud náročnější, ale zato dosahuje



až překvapivě dobré rychlosti, úlohu můžeme vyřešit v čase $O(NM)$. Podrobnou analýzu tohoto řešení včetně programu v Pascalu najdete v [14].

## CVIČENÍ

**1.** Tzv. Hammingova posloupnost je vzestupně uspořádaná posloupnost kladných celých čísel, která nejsou dělitelná jinými prvočísly než 2, 3 a 5. Posloupnost začíná čísly 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, ... Sestavte algoritmus, který pro dané $N$ určí prvních $N$ členů Hammingovy posloupnosti. *Návod*: Použijte podobný postup jako v př. 1, kap. 15.2.

*Pozn.*: Různé varianty řešení velmi podobné úlohy naleznete v [14].

**2.** Je dána posloupnost celých čísel. Její délka není předem známa, posloupnost může být velmi dlouhá. Určete v ní délku co nejdelšího souvislého úseku, v němž se libovolná dvě čísla liší maximálně o danou hodnotu $D$. Např. pro posloupnost 5 7 6 7 7 8 8 7 9 9 9 a $D = 2$ bude výsledkem 8 (což je délka úseku 7 7 8 8 7 9 9 9). *Návod*: Zobecněte řešení př. 2 z kap. 15.3, úloha má řešení s lineární časovou složitostí.

**3.** Na kružnici je rozmístěno $N$ bodů tak, že všechny jejich vzájemné vzdálenosti měřené po obvodu kružnice jsou celočíselné. Jsou dány vzájemné vzdálenosti všech dvojic bodů, které spolu na obvodu kružnice sousedí. Zjistěte, zda mezi danými $N$ body existují čtyři takové, které leží ve vrcholech čtverce. Pokud ano, jednu takovou čtveřici bodů nalezněte. *Návod*: Zobecněte řešení př. 3 z kap. 15.3, úloha má řešení s lineární časovou složitostí.

*Pozn.*: Různé varianty řešení této úlohy naleznete v knize [14].

**4.** Je dána obdélníková matice celých čísel velikosti $N \times M$. Nalezněte v ní podmatici (tj. souvislý obdélníkový výřez) s co největším součtem prvků. Řešením úlohy je tento maximální součet. Nejlepší známé řešení má časovou složitost $O(NM \min(N, M))$.

**5.** Upravte řešení př. 2 z kap. 15.4 tak, jak je naznačeno v textu. Program bude po úpravě vybírat výhodnější směr předvýpočtu a jako výsledek bude dávat také polohu nalezené maximální jedničkové podmatice.

## SEZNAM PROGRAMOVÝCH UKÁZEK

V závorce je u každé programové ukázky uvedeno číslo kapitoly, kde se v textu knihy program nachází, a jméno souboru na disketě, v němž je odladěný program uložen.

Rozklad čísla na součet dvou třetích mocnin (15.2, ROZKLAD.PAS)
Nalezení úseku s maximálním součtem prvků (15.3, MAXSOUC.PAS)
Nalezení nejdelšího „hladkého" úseku v posloupnosti (15.3, MAXHLAD.PAS)
Nalezení dvou bodů ležících na průměru kružnice (15.3, BODYKRUZ.PAS)
Určení velikosti maximální podmatice ze samých jedniček (15.4, JEDNICKY PAS)



## LITERATURA


[1] BENTLEY JON: *Programming Pearls*, Addison-Wesley, 1986, slov. překlad: *Perly programovania*, Alfa Bratislava, 1992

[2] DEMEL J.: *Grafy*, SNTL Praha, 1988

[3] DIJKSTRA W. EDSGER: *A Discipline of Programming*, Prentice-Hall Inc., 1976, rus. překlad: *Disciplina programmirovanija*, Mir Moskva, 1978.

[4] DIJKSTRA W. EDSGER, FEIJEN W.H.J.: *A Method of Programming*, Addison-Wesley Inc., 1988

[5] DRÓZD JANUŠ, KRYL RUDOLF: *Začínáme s programováním*, Grada Praha, 1993

[6] GRIES DAVID: *The Science of Programming*, Springer Verlag, 1981, rus. překlad: *Nauka programmirovanija*, Mir Moskva, 1984

[7] HONZÍK J. A KOL.: *Programovací techniky*, AK Slušovice, 1986

[8] HUDEC B.: *Programovací techniky*, ČVUT Praha, 1989

[9] CHYTIL MICHAL, KŘIVÁNEK MIRKO: *Výpočtová složitost*, MFF UK Praha, 1988

[10] JINOCH JOSEF, MÜLLER KAREL, VOGEL JOSEF: *Programování v jazyku Pascal*, SNTL Praha, 1987

[11] KNUTH D. E.: *The Art of Computer Programming, Vol. 1.: Fundamental Algorithms*, Addison-Wesley, 1968, rus. překlad: Mir Moskva, 1977

[12] KNUTH D. E.: *The Art of Computer Programming, Vol. 3.: Sorting and Searching*, Reading, Addison-Wesley, 1973, rus. překlad: Mir Moskva, 1978

[13] KUČERA LUDĚK: *Kombinatorické algoritmy*, SNTL Praha, 1989

[14] LIBICHER IVAN, TÖPFER PAVEL: *Od problému k algoritmu a programu*, Grada Praha, 1992

[15] NEŠETŘIL JAROSLAV: *Teorie grafů*, SNTL Praha, 1979

[16] RYCHLÍK JAN: *Programovací techniky*, KOPP České Budějovice, 1992

[17] TÖPFEROVÁ DANA, TÖPFER PAVEL: *Sbírka úloh z programování*, Grada Praha, 1992

[18] WINCZER M.: *Korešpondenčný seminár z programovania 1983 až 1988*, MFF UK Bratislava, 1993

[19] WIRTH NIKLAUS: *Algorithms + Data Structures = Programs*, Prentice-Hall Inc., 1975, slov. překlad: *Algoritmy a štruktúry údajov*, Alfa Bratislava, 1988

Výše uvedený seznam literatury je dosti různorodý. Obsahuje jak elementární učebnice pro začátečníky, tak cizojazyčné odborné publikace. Pokud s programováním začínáte, je pro vás určena základní učebnice programování v Pascalu [5]. Základní úlohy na procvičování můžete čerpat ze sbírky [17], trochu náročnější příklady naleznete třeba v publikaci [18]. V obou posledně jmenovaných knihách vám při práci pomohou stručné návody k řešení úloh.

Pokročilejší čtenáři se mnohé o algoritmech a programování dozvědí z klasické učebnice programování [19]. Ta je z celého seznamu literatury obsahově nejbližší této knize, je však místy poněkud náročnější na čtení. Oproti naší knize je v [19] podrobněji zpracována například problematika třídění a dynamických datových struktur, na druhé straně některé zde uvedené algoritmy a postupy tam nejsou vůbec obsaženy.

Algoritmům a programovacím technikám bylo věnováno již i několik původních českých publikací. Konkrétně se jedná o dvoje učební texty [7] a [8] a o knihu [16]. Obsah všech tří uvedených příruček se částečně překrývá. Ačkoliv se všechny tři zmíněné texty jmenují shodně „Programovací techniky", pojednávají více o konkrétních známých algoritmech než o obecných technikách návrhu efektivních algoritmů. Naše kniha přináší rozsáhlejší a ucelenější pohled na problematiku návrhu algoritmů s větším důrazem na kvalitu vytvářených programů. Zejména knihu [16], která je ze všech tří uvedených publikací nejnovější a nejdostupnější, však můžeme čtenářům rozhodně doporučit. Oproti naší knize v ní můžete nalézt zejména podrobnější zpracování některých algoritmů pracujících se stromy (konkrétně vyvážené AVL-stromy a obecné B-stromy).

Skripta postgraduálního kurzu [9] podávají celkový přehled o problematice složitosti algoritmů. Hlubší pohled na návrh algoritmů a jejich posuzování z hlediska efektivity naleznete v knize [14], a to ve formě ukázkově řešených úloh. Metodám efektivního řešení programátorských úloh je věnována také kniha [1].

Monografie [11] a [12] přinášejí podrobný přehled a rozbor řady používaných algoritmů. Celá kniha [12] je věnována problematice třídění



a vyhledávaní dat. Mnoho potřebných a užitečných algoritmů z oblasti kombinatoriky a teorie grafů naleznete i v českých knihách [2], [13] a [15].

Knihy [3], [4] a [6] jsou určeny zkušenějším čtenářům. Jsou to odborné publikace věnované obecně programování jakožto vědnímu oboru, metodice tvorby algoritmů a programů a principům dokazování jejich správnosti. Začátečníkovi by však mohl připadat způsob vyjadřování autorů a použitý matematický formalismus poněkud nečitelný a nesrozumitelný.

Pokud se budete chtít blíže seznámit s některým konkrétním programovacím jazykem, abyste si získané znalosti mohli vyzkoušet ve své vlastní programátorské praxi, vhodnou učebnici si jistě najdete sami. Programovací jazyk Pascal používaný v této knize k zápisu programových ukázek se můžete naučit například z tradiční příručky [10]. Její nevýhodou je to, že vás seznámí pouze se standardním Pascalem podle normy. Při práci na počítači budete používat některou implementaci jazyka (například Turbo Pascal od firmy Borland při práci na počítačích typu PC), která se může od normy v některých věcech odlišovat. Musíte se proto seznámit také s konkrétním produktem a verzí, kterou chcete používat. Na našem trhu existuje více různých učebnic Pascalu a najdete i česky psané učebnice Turbo Pascalu.

## REJSTŘÍK

Některé víceslovné pojmy jsou v rejstříku zařazeny dvakrát, abecedně podle počátečního slova a podle podstatného jména (např. binární strom, strom binární). Rejstřík se tím sice trochu prodloužil, ale zato se vám v něm bude snáze vyhledávat.

RNDr. Pavel Töpfer, CSc.

# ALGORITMY
## a programovací techniky

Obálku navrhl Karel Horák
Vydalo nakladatelství PROMETHEUS, s.r.o.,
Žitná 25, 117 10 Praha 1,
roku 1995 jako svou publikaci č. 150
Odpovědná redaktorka RNDr. Jana Vlášková
Sazbu programem TEX připravil Karel Horák
Vytiskl Pragopress, Václavská 12, Praha 2
1. vydání

**ISBN 80-85849-83-6**